Scheduling problems in automated manufacturing

Acknowledgements

Late in the evening of January 31 1992, I moved into my new apartment in Maastricht. Early in the morning of February 1 1992, I had my first meeting with Yves Crama, my thesis advisor. I took from my bag, in which I had put everything that I needed before 9.00 AM that morning, the working papers I had read, and put them on Yves' desk. While I concentrated on keeping my eyes open, Yves noticed that my shampoo, which I also had needed before 9.00 AM, had mingled with the papers, and was now conquering his desk. 'Wat is dit?', he asked, (which is probably best translated as, 'What is this?'), as he started cleaning his desk.

And that was just the beginning. For four years I kept spreading combinations of research output and rubbish on his desk, and poor Yves kept cleaning. But now, after four years of cleaning, I am finally able to answer his hypothetical question. It was the beginning of a Ph.D. Thesis. I am well aware, however, that this belated answer would have been quite different if our teamwork had been less smooth. I owe a lot to Yves for his adequate, wise, and always stimulating responses to whatever I put on his desk.

None of all this would have happened, if Antoon Kolen had not offered me the opportunity to work in Maastricht for four years. Throughout this period, he succeeded in creating a very friendly, relaxed and stimulating research environment. Over the past four years, he switched succesfully between the roles of an encouraging boss, a friendly colleague, a bridge partner and opponent. Antoon has brought together a group of people who I have become to appreciate as colleagues and, more importantly, as friends.

During the first three years, I shared my office with Jaap Geerdink. Jaap has been a great companion in these first years, both when working and when socializing. It has been a pleasure to learn how to appropriately anticipate his cautious manners. Olaf Flippo arrived in Maastricht in the fall of 1992. Through the years, I have shamelessly made use of his unboostably good temper and his willingness to spell out such boring things as research papers or a Ph.D. Thesis. Stan van Hoesel arrived in the summer of 1994. His interest in combinatorial optimization and its researchers has been inspiring and enjoyable. Thanks gentlemen, for your time and support.

In the spring of 1995, Eugene Levner established e-mail contact between Maastricht and Jerusalem. Since then, we had many discussions on robotic cell scheduling problems. Chapter 2 is based on these discussions. Thanks Eugene, not only has each of your mail messages been a pleasure to receive and read, this thesis has benefitted from their constructive and cooperative content.

Apart from Yves Crama, who has actively coached and co-authored most of the chapters of this thesis, Chapters 5 and 6 are also co-authored by Frits Spieksma and Olaf Flippo. It has been a pleasure to write and discuss together the research described in these chapters. These chapters would not have been written without the cooperation

of Mr. Voogt en Mr Driessen of the Center For Manufacturing Technology of Philips Nederland N.V.. Many thanks for providing us with the problem studied in Chapter 6, and the data. Thanks also to Jaap, Olaf, Marcel Canoy and Giorgio Gallo, and several anonymous referees for helpful discussions and comments on earlier versions of some parts of this thesis.

Graphical support was provided by Olaf, Stan, Jaap, and, most of all, Rob Pauly. Thank you all. Many thanks also to my irregular roommates Ron van der Wal, in particular for his patient attempts to teach an Operations Research group something as complicated as programming in C++, and Hugo Kruiniger, for our amusing discussions on all other subjects. Thanks also to the other members of the department of Quantitative Economics, especially Karin, Yolanda, Miranda and Ellen. Thanks to Robert, Arie and Maarten for pleasant and inspiring discussions on various combinatorial topics and alike. Thanks also to Jan Nijhuis, for his interest, and for working together on the Productions Management courses. I know at least one person who learnt a lot from these courses. Many thanks to Shell Nederland N.V. for sponsoring my trip to the Mathematical Programming Symposium in Ann Arbor, Michigan.

Scheduling Problems in Automated Manufacturing

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Rijksuniversiteit Limburg te Maastricht, op gezag van de Rector Magnificus,
Prof. Mr. M.J. Cohen, volgens het besluit van het College van Dekanen,
in het openbaar te verdedigen op vrijdag 7 juni 1996 om 14.00 uur.

door

Joël Joris van de Klundert

Promotores:

Prof. dr Y. Crama (Université de Liège)
Prof. dr ir A.W.J. Kolen

Beoordelingscommissie:

Prof. dr ir drs O.J. Vrieze (voorzitter)
Prof. dr J. Blazewicz (Instytut Informatyki, Poznan, Poland)
Prof. dr J.K. Lenstra (Technische Universiteit Eindhoven)

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Klundert, Joël Joris van de

Scheduling problems in automated manufacturing
Joël Joris van de Klundert. - Maastricht : Unigraphic. Ill.
Proefschrift Rijksuniversiteit Limburg, Maastricht.
NUGI 811/687.
ISBN 90-5681-006-05

Contents

1 Introduction and overview 5
- 1.1 Scheduling problems in automated manufacturing 5
- 1.2 Robotic cells 8
- 1.3 Printed circuit board assembly 10
- 1.4 Tool management 11

I Robotic cells 13

2 Scheduling problems in robotic cells 15
- 2.1 Introduction 15
- 2.2 Robotic cells 17
  - 2.2.1 Basic ingredients 17
  - 2.2.2 Processing requirements 18
  - 2.2.3 Time models for the robot 19
  - 2.2.4 Problem size 19
  - 2.2.5 Objective functions 20
  - 2.2.6 Related problems 21
- 2.3 Optimization problems and models 21
  - 2.3.1 Processing windows and identical parts 26
  - 2.3.2 No-wait time windows and identical parts 30
  - 2.3.3 Unbounded time windows 31
- 2.4 Cycle time computation 32
  - 2.4.1 Time windows and identical parts 32
  - 2.4.2 Unbounded time windows 32
- 2.5 Other topics and further research 38

3 Scheduling of identical parts in a robotic flowshop 41
- 3.1 Introduction 41
- 3.2 Cycles, permutations & schedules 44
- 3.3 Pyramidal permutations. 46
- 3.4 An algorithm for computing the cycle time of a pyramidal permutation. 55
- 3.5 Polynomial algorithms for the identical parts cyclic scheduling problem 59

3.6 Summary and directions for further research 67

4 The optimality of short robot move sequences in a robotic flowshop 69
4.1 Introduction 69
4.2 A state space graph model 69
4.3 Results 74
4.4 Generalisations and further research 83

II Printed circuit board assembly 85

5 The component retrieval problem in printed circuit board assembly 87
5.1 Introduction 87
5.2 The Fuji CP II placement machine 88
5.3 The Component retrieval problem as a PERT/CPM network model with design aspects 90
5.4 An example why straightforward dynamic programming does not work 93
5.5 A polynomial algorithm for CRP 95
5.5.1 A simplified version of CRP 96
5.5.2 Further properties of the s-labels 99
5.5.3 The general case 102
5.5.4 Example 104
5.6 An *NP*-hard generalization of CRP. 106
5.7 Conclusions 110

6 A case study in printed circuit board assembly 111
6.1 Introduction 111
6.2 Problem description 114
6.2.1 Properties of the assembly environment 114
6.2.2 The placement machine 116
6.2.3 The component retrieval problem 117
6.3 The planning procedure 119
6.3.1 Constructing a feeder rack assignment: Phase 1 121
6.3.2 Constructing a component placement sequence: Phase 2 127
6.4 Computational results. 128
6.5 Conclusions 132

III Tool management 133

7 The approximability of tool management problems 135
7.1 Introduction 135
7.2 Models and complexity 136
7.3 Approximation algorithms and worst case ratios 139

7.4 Further research 150

8 Approximation algorithms for integer covering problems via greedy column generation 151
8.1 Introduction 151
8.2 αGreedy algorithms and their worst case ratio. 154
8.3 HyperαGreedy algorithms and their worst case ratio. 160
8.4 Applications 166

Bibliography 169

Samenvatting 179

Curriculum vitae 183

Chapter 1

Introduction and overview

1.1 Scheduling problems in automated manufacturing

The subject of this thesis is scheduling in automated manufacturing. Let us start by explaining how this subject is to be understood, and why it is of interest.

The process of technical innovation and automation, together with the ever fiercer competition for customers, has led to dramatic changes in production and production management since the early sixties. Today's manufacturers are producing a much wider variety of products than the manufacturers of the 1950's, while product life cycles have become much shorter. Also, in many industries the globalisation of market competition has put high demands on the efficiency of production. In traditional manufacturing systems, however, the high production volumes needed to achieve low costs do not combine well with the aforementioned product variety and short life cycles. Flexible manufacturing systems strive to combine low costs with medium volume production of a wide variety of products. Furthermore, the introduction of new products is relatively easy in flexible manufacturing systems. As such, flexible manufacturing systems can provide (partial) solutions to many of the problems that today's manufacturers are facing. Let us therefore first take a closer look at these flexible manufacturing systems and see how they can achieve the aforementioned benefits.

'A flexible manufacturing system (FMS) is an integrated, computer-controlled complex of automated material handling devices and numerically controlled machine tools that can simultaneously process medium-sized volumes of a variety of part types', Stecke [1983]. Systems that fit this definition have been installed in many important industries such as metal working and cutting industries. In this thesis we use the terms automated manufacturing system and flexible manufacturing system interchangeably. We distinguish the following constituents of such systems:

1. *Flexible machines*, (computer) numerically controlled machines that are able to produce a wide variety of parts, possibly in several ways, using a set of tools.

2. *Tools*, that can reside in a central storage area, be transported, reside in the tool magazine of a machine, or be in use by a machine. Machines are able to switch fast from using one tool to another, provided that the required tools are in the tool magazine of the machine. If, however, tools are needed that have to be loaded onto the tool magazine first, (perhaps by some automated tool handling system,) set up times are significantly larger.

3. An *automated material handling system* that transports parts between machines or between machines and a storage area, and may also perform other activities such as loading and unloading of parts onto and from machines and/or pallets. The material handling system may consist of several devices.

4. *Central inventory storage area(s)* for raws, spare parts, work in process and/or finished goods. In many FMSs, storage areas are also highly automated.

Due to the tooling facilities, the small set up times (which result from small tool switch over time), and their automated control and coordination, automated manufacturing systems can achieve the low cost that used to be associated with high volume production, while producing only medium-sized volumes. For numerically controlled machines, performing operations comes down to automatically executing a set of instructions, e.g. a computer program. The versatility of the machines that results from their tool hand-ling abilities, and their (programmable) automated control makes it relatively easy to introduce new products into the system.

Several authors (see Crama [1995] for references) have investigated what it means exactly for a manufacturing system to be *flexible*. Many different types of flexibilities have been defined, but among the more important ones we mention:

1. *market flexibility*, the abililty of the system to adapt its product mix to changes in demand, e.g. by changing the production volumes or even introducing new products;

2. *machine flexiblity*, the ability of machines to perform various operations without requiring a prohibitive effort in switching from one operation to another;

3. *material handling flexibility*, the ability of the material handling system to move different parts efficiently for proper positioning and processing through the system;

4. *operation flexibility*, the ability of a part to be produced in several ways;

5. *routing flexibility*, the ability of the system to produce a part by alternate routes through the system.

Although flexible manufacturing systems may have many attractive characteristics, it is far from easy to manage and operate such systems so as to fully utilize their possibilities. As a result, not all installed FMSs have been equally succesful. Jaikumar

[1985] for instance, reports large variations in the flexibility achieved by FMS users. He also claims that 'Rather than narrowing the competitive gap with Japan, the technology of automation is widening it further'.

One of the management tasks of operating manufacturing facilities whose complexity has increased significantly due to the introducion of FMSs, is production planning and scheduling. Stecke [1983] points out that there is a need for planning procedures that tackle the increased complexity of these planning problems. The seemingly infinite number of planning alternatives as well as their complexity make it hard to assess their relative quality, and therefore to select a good alternative. Several authors (see e.g. Stecke [1983], Jaikumar and Van Wassenhove [1989]) have proposed and developed planning procedures of a hierarchical nature, which they claim to be quite successful in practice. However, particularly at the lower levels of medium and short range planning, several complex problems may pop up that are not encountered in production planning models for more classical manufacturing systems. In FMSs, medium and short range scheduling decisions consist for example in assigning parts and operations to machines, in planning of material handling activities, and in planning tool switches and other tool handling decisions, over a time horizon ranging from several hours to one or two months. Of course, all these decisions should be consistent with one another.

For more classical production situations, the analysis of the scheduling problems arising in such situations has lead to the development of a 'theory of scheduling', and many insights obtained from this theory have been successfully applied to the solution of real world problems. Blazewicz et al. [1993] propose the following definition of scheduling problems : 'scheduling problems deal with the allocation of scarce resources over time to perform a set of tasks'. In general, scheduling is a rich and active research area, and in particular machine scheduling problems, in which a set of parts has to be processed on one or more machines, have received considerable attention. Still, most of the scheduling models that deal with production and/or machine scheduling apply to classical production settings and do not incorporate several of the important characteristics of state of the art automated manufacturing systems. In order to utilize the full power of flexible manufacturing systems however, it is particularly important to handle and schedule well the attributes to which their flexibility should be contributed. This has recently led many researchers to look into such problems as the tooling and material handling problems that arise in automated manufacturing.

We conclude that in many industries, the effective and efficient management of automated manufacturing systems has become a necessity in order to stay in tune with today's leading manufacturers. One of these management tasks is medium and short range planning and scheduling. Mostly, these scheduling problems, and especially those addressing the features of automated systems that distinguish them from more conventional ones, are very hard to solve. In addition, existing scheduling theory does not provide answers to several of the newly posed scheduling problems in automated manufacturing. So, it becomes clear that there is a need to investigate scheduling models in automated manufacturing, and that these scheduling problems offer many challenges.

Many of the scheduling problems in such an automatically controlled environment have a deterministic character. Such deterministic scheduling problems can often be modelled by combinatorial models that are akin to well known models and problems, and for which software implementions of combinatorial optimization techniques have proven to be successful solution methods. The automated nature of flexible manufacturing systems and specifically their automated control make them an ideal environment for applying such automated solution procedures. Hence, one could reasonably expect that the tool kit of combinatorial optimization can assist in solving scheduling problems arising in automated manufacturing. This thesis investigates combinatorial models and solution methods for deterministic scheduling problems that arise in automated manufacturing.

There are numerous deterministic scheduling problems arising in automated manufacturing of which this thesis addresses only a few. Being aware of the fact that both scheduling theory and the practice of automated production planning are important research areas, which should stimulate each other, we have attempted to select and analyze several interesting and important problems that contribute to both these fields. Hence, the research presented in this thesis ranges from mathematical analysis of rather abstract models, mainly focusing on the complexity of solving them exactly or approximately, to (mathematical) analysis of several models for specific real life production settings, and computational studies in solving real life problems. The manufacturing environments we study feature varying degrees of automation. However, all problems we consider possess characteristics that make them distinct from more classical scheduling problems. We address various issues, such as scheduling of flexible machines, tool management problems, scheduling of material handling devices, and scheduling problems in which the interaction between several attributes of FMSs is of crucial importance.

Each of the following three sections of this chapter overview the research presented in one of the three parts of this thesis. Part I, which consists of Chapters 2, 3 and 4, addresses robotic cells. Part II, made of Chapters 5 and 6, concerns printed circuit board assembly. Part III, which consists of Chapters 7 and 8, studies tool management problems. In order to improve the readability of this thesis, we have tried to write Chapters 2 to 8 so that they can be read independently from each other. This implies that some of the chapters contain material that is redundant for those readers who do the author the favor of reading more than one chapter.

1.2 Robotic cells

Medium range production planning for automated manufacturing systems often requires to select a production plan from an endless variety of plans whose analysis is a cumbersome task. This variety is reduced when machines are divided into groups, for example by applying the widespread concept of group technology. The idea is then to

also partition the parts into groups, such that all parts in a part group can be processed completely by the machines in one machine group. This partitioning can lead to a reduction in the complexity and size of the various scheduling problems. Furthermore, this grouping is likely to cause a reduction in material handling activities, and the problem of scheduling these activities may become easier to tackle. Such groups of flexible machines are often refered to as flexible manufacturing cells. Part I of this thesis, i.e. Chapters 2, 3 and 4, addresses scheduling problems arising in such cells in which the material handling is performed by a single device, be it a robot, an automatic guided vehicle (AGV), or a hoist. If the material handling is performed by a robot, such cells are usually called robotic cells.

Chapter 2 addresses several problems and models arising in robotic cells and more specifically, robotic flowshops. A typical scheduling problem in such cells is as follows. There are a number of machines, an input device and an output device. A certain set of parts is to be repeatedly produced. Each part requires processing on every machine and the order in which the parts have to visit the machines is fixed and identical for all parts. Initially, parts are available at the input device and, when completely processed, they have to be delivered at the output device. The transportation of the parts between the machines and the devices, and the loading and unloading of the parts on the machines and these devices is performed by a single robot or AGV. In general, the problem is then to specify two sequences : a part input sequence, that is an order in which the parts are to be taken from the input device, and a robot move sequence, that is an order in which the robot is to execute its activities. In Chapter 2 we overview the literature on scheduling problems arising in robotic cells. One of our aims in Chapter 2 is to establish links between the research of various authors (e.g. Sethi et al. [1992], Lei & Wang [1991], Levner, Kats & Meyzin [1995]) on scheduling problems in robotic cells, and to establish links between this stream of research and other topics of combinatorial optimization. In this process, we identify several open problems and we establish several new results on these problems. We give an NP-completeness proof for a problem that is general enough to contain many of the problems considered to date as a special case. In this problem the processing requirements are specified by means of so called processing windows (i.e. lower and upper bounds on processing times). We overview research conducted on special cases of this general problem. We also present an exact polynomial time algorithm to compute the cycle time when both the part input sequence and the robot move sequence are given, and the processing windows reduce to well-defined processing times. This algorithm is more general than several algorithms previously presented by Ioachim & Soumis [1995] and Hall et al. [1995], and its computational complexity is lower.

Chapter 3 considers a special case of our general problem in which all parts have the same processing requirements, and these processing requirements are again simply stated by means of processing times. We show that among the 1-unit cycles (i.e., cyclic robot move sequences with the property that exactly one part is processed in each cycle), there always exists an optimal robot move sequence that has a so called pyramidal structure. Pyramidal sequences have also been investigated in relation to

the Traveling Salesman Problem. For the TSP, a shortest pyramidal tour can be found in polynomial time using dynamic programming techniques. Similarly, we present two polynomial time dynamic programming algorithms to compute the optimal robot move sequence in such robotic cells. The complexity of both algorithms depends polynomially on the number of machines. One of these algorithms is strongly polynomial.

In Chapter 3, we optimize only over a subset of the set of all possible robot move cycles, i.e. the subset consisting of all 1-unit cycles. For cells with 3 machines, this subset has been conjectured by Sethi et al. [1992] to always contain an optimal solution. Some evidence for this conjecture can be found in Hall et al. [1995a]. In Chapter 4, we show that a slightly weaker version of the conjecture holds for three-machine cells.

1.3 Printed circuit board assembly

Advances in technology have led to the use of highly automated machines for the assembly of printed circuit boards. In Chapters 5 and 6, which form the second part of this thesis, we consider scheduling problems that arise in this context. One of the more time consuming phases in the process of printed circuit board (PCB) assembly is the placement of components on bare PCBs by so called component placement machines. These machines have a device, called feeder rack, to hold feeders (tapes) of components. Each feeder contains components of a same type. These machines retrieve components from the feeders and place them on the board. In this framework, single machine scheduling problems require to specify a component placement sequence and to specify an assignment of tapes to positions in the feeder rack. Typically, the assembly of each PCB involves several machines. Hence, one can also consider the problem of assigning PCBs to machines, and assigning placement operations of the boards to machines, so as to maximize some production performance measure, e.g. to balance workload.

This context has many similarities with automated manufacturing systems encountered in metal cutting and other industries. For instance in the computational study conducted in Chapter 6, we encounter a flowshop like environment. Moreover, the assignment of components to machines is closely related to the tool management problems discussed in Chapters 7 and 8.

In Chapter 5 we investigate, for a commonly used machine, the minimization of the makespan of the assembly of a single PCB when the component placement sequence and the feeder-rack assignment are fixed. The only remaining decision in this problem is to decide from which feeder to pick components, when several feeders hold components of a same type. We show in Chapter 5 that this task is not as easy as claimed by Bard, Clayton and Feo [1994], but can still be solved in polynomial time, using dynamic programming techniques. For this purpose, we reformulate the problem as a graph problem that is interesting in its own right. We analyse the complexity of a slight generalization of this problem, and show that it is NP-Complete.

Chapter 6 describes a computational study in PCB manufacturing on real life data made available to us by the Center For Manufacturing Technology of Philips Neder-

land N.V.. In this study we are asked to balance a flowshop of component placement machines, on which a family of different boards is to be produced. This requires to specify an assignment of tapes to feeder rack positions of the machines, which enables short component placement sequences for all boards on every machine. The approach taken by Philips is to map the family of boards onto a single composite board and to balance out the flowshop for this single composite board. We try a different route, and improve significantly over their results by accounting as much as possible for individual board characteristics. We reduce the sum (over all boards) of the maximum makespans (over all machines) on average by 17 % and show that improvements of more than 25 % are impossible to attain.

1.4 Tool management

The ability of flexible machines to use various tools in order to perform a variety of operations, is one of the most fundamental characteristics of flexible manufacturing systems. Gray, Seidmann & Stecke [1993] claim that 'it is clear that a lack of attention to structured tool management has resulted in the poor performance of many manufacturing systems.' Moreover, they report that tooling may account for up to 30 % of total fixed and variable costs of production. Indeed, tooling decisions play a role in many of the scheduling problems that arise in the scheduling of FMSs and have received considerable attention in the literature. Part III of this thesis, consisting of Chapters 7 and 8, studies tool management problems.

As stated in the first section of this chapter, tool handling operations may become time consuming when tools are switched between the tool magazines of a machine and an external device, e.g. a tool storage area. Such switches may be unavoidable because of limited tool magazine capacity. Given a set of jobs to be produced on a set of machines, one could therefore consider the problem of finding a production plan that minimizes the time spent on switching tools. If tool switches are necessarily performed sequentially, this may translate to minimizing the number of tool switches (Tang & Denardo [1988a]). On the other hand, if tools can be switched simultaneously, minimizing the number of switching moments, or switching instants, is a natural objective (Tang & Denardo [1988b]).

Unfortunately, even the most basic tooling problems, which are a built-in component of many more general problems, turn out to be very hard to solve (see e.g. Crama, Oerlemans & Spieksma [1995]). We define a batch or group as a set of jobs that can be processed without tool switches. The problem of finding a largest batch is called the Batch Selection Problem. This problem is strongly NP-hard. The problem of partitioning the set of jobs into a minimum number of batches is called the Job Grouping Problem, and is also known to be strongly NP-hard. In Chapter 7 we investigate the complexity of finding approximate solutions of guaranteed quality for these problems. We study the worst case ratios of polynomial time approximation algorithms as they have been proposed to date. All of these algorithms turn out to exhibit rather poor

worst case behavior. We present several results addressing the approximability of these problems, and the interrelationships between their approximability.

Most algorithms for the Job Grouping problem generate batches by solving iteratively the Batch Selection problem. In Chapter 8 we present results on the worst case ratio of the resulting algorithm for Job Grouping expressed in terms of the worst case ratio of the approximation algorithm used to solve the Batch Selection problem. Our results are general enough to apply to a broad class of pairs of combinatorial problems that are related in a similar ('master-slave') fashion.

Part I

Robotic cells

Chapter 2

Scheduling problems in robotic cells

2.1 Introduction

Over the last two decades, the automation of production technology, in combination with advances in production planning, has dramatically changed the equipment used by manufacturing companies and the way production is organized. Together, they have led to an enormous increase in efficiency and flexibility. Progress in terms of automation has therefore become a necessity to survive the fierce global competition. As a consequence, highly automated or even unmanned manufacturing systems have become common in several industries. The resulting need for new, effective, control and scheduling systems is widely recognized by practitioners as well as researchers.

Typically, automated manufacturing involves intelligent, flexible, machines, that can be programmed to produce a variety of parts with little or no setup cost. Often, these flexible machines are grouped into cells, in such a way that the entire production of each part can be performed in one of these flexible cells. One of the benefits of machine pooling is the reduction in material handling activities that it entails. Within a cell, material handling is usually performed by one (or a few) robots or Automatic Guided Vehicles (AGVs). When this is the case, the performance of the cell becomes highly dependent on the interaction between the automated material handling device(s) and the machines.

The relatively small number of machines and material handling devices in flexible cells, as well as their high degree of automation, make them an ideal environment for automated production scheduling. As a matter of fact, in several industrial applications, the use of advanced planning software has been reported to improve substantially the performance of the cells. Classical scheduling models however, as they have been developed until the late seventies, appear to be unsuitable to incorporate the most important characteristics of flexible manufacturing cells, such as the interaction between the material handling system and the machines. Hence, a diverse lot of new and challenging scheduling problems, arising in the context of manufacturing cells, has re-

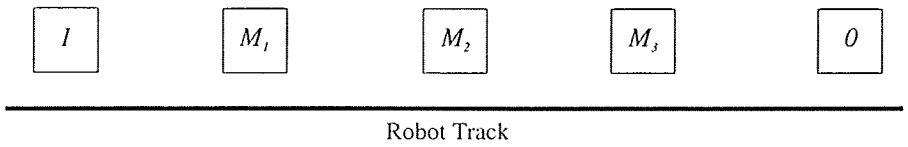

Figure 2.1: A 3-machine robotic cell (line layout)

cently appeared in the literature. In this chapter, we attempt to take a systematic view on those problems that specifically deal with the aforementioned interaction between the material handling device and the machines. In particular, we focus on a class of problems known as *robotic cell scheduling problems.*

Although most of the research on this topic is quite recent, a wide variety of robotic cell scheduling problems have been investigated to date. In this chapter, we classify and overview models, problems, algorithms and complexity results mentioned in the literature and discuss their interrelationships. Our emphasis is on constructive and exact results in the realm of deterministic scheduling, rather than empirical or simulations studies. Our contribution lies in establishing and discussing the commonalities and differences between papers that largely ignore one another, and in showing their relationship with other fields of investigation (e.g. project scheduling). En route, we improve some of the results published in the literature and identify several open problems, for which we are sometimes able to provide (partial) solutions.

In Section 1, we describe various robotic cell layouts and scheduling problems as they have appeared in the literature. This section does not attempt to give an exhaustive overview of the area of robotic cell scheduling. As a matter of fact, as this research area is moving and expanding rapidly, we do not even claim to give a complete overview of the topics that we study at length in subsequent sections. We simply try to pay some attention to cells and problems which we believe to be particularly important and interesting. Our main focus is on robotic flowshop scheduling problems without buffers (see Figures 2.1 and 2.2). (This emphasis on bufferless systems is consistent with modern production philosophies such as Just In Time or lean manufacturing.)

A *robotic flowshop* consists of m machines $M_1, \ldots, M_m$, an input device I or M_0, an output device O or M_{m+1}, and a robot. The robot performs all material handling operations in the cell, i.e. the transportation of parts between the machines and the devices, and the loading and unloading of the parts onto and from the machines and devices. All parts are initially available at the input device, and must be sequentially processed on M_1, $M_2, \ldots, M_m$, until they are finally unloaded from M_m and delivered at the output device.

A most general optimization problem for robotic flowshops asks to specify a *sequence of robot moves* and a *part input sequence* (i.e. the order in which the parts are to be taken from the input device) so as to maximize a predetermined production performance measure (for instance, the throughput rate of the cell).

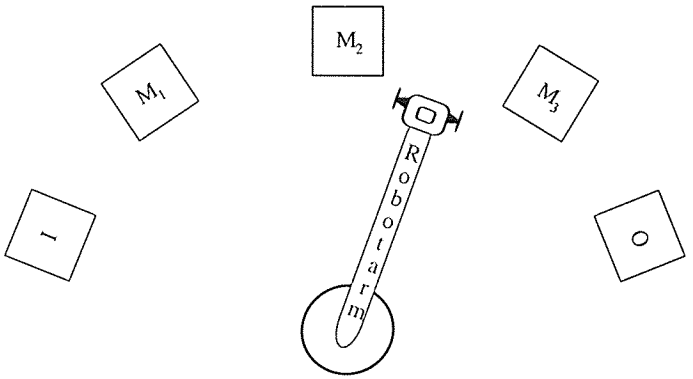

Figure 2.2: A 3-machine robotic cell (circle layout)

In Section 3, we discuss the complexity of several variants of this double sequencing problem. A number of these variants, in which one of the two sequencing problems has vanished or is treated as given, turn out to be particulary interesting. Section 4 deals with problems that arise when both sequences are treated as given. More precisely, we discuss the problem of computing the optimal cycle time of a production plan in which both the robot move sequence and the part input sequence are given. We improve and/or generalize several published results. In Section 5, we discuss other aspects of robotic cell scheduling problems and we suggest a number of new research directions.

2.2 Robotic cells

In this section, we survey a variety of robotic cell models and the corresponding basic scheduling problems. The bufferless robotic flowshop provides the starting point of our overview. Later on, we also consider problems arising in different cell layouts, e.g. cells with buffers or parallel machines.

2.2.1 Basic ingredients

Let us first collect the constituents of the scheduling problem for bufferless robotic flowshops:

1. m machines $M_1, \ldots, M_m$.

2. An input device, denoted I or M_0, and an output device denoted O or M_{m+1}.

3. A set of parts J. Part $j \in J$ has processing requirement r_i^j on machine $M_i, i = 1, \ldots, m$. The parts are initially available at the input device, must be processed

on every machine, in increasing order of the indices of the machines, and must finally be delivered at the output device.

4. A robot that performs the transportation of parts between the machines. The loading and unloading of parts onto or from the machines also requires the attendance of the robot.

5. There are no buffers in the flowshop, and the machines as well as the robot can only process one part at a time. In particular, if a part is between the input and output device, then it is either on a machine, or being carried by the robot.

In general terms, the robotic flowshop scheduling problem is to determine an ordering of the parts at the input device (a *part input sequence*), a sequence of the robot activities (a *robot move sequence*) and the start times and finish times of these activities (a *schedule*) so as to optimize production performance.

We shall see that, in many cases, an optimal schedule can be efficiently computed once the part input sequence and the robot move sequence are known. For this reason, the robotic cell scheduling problem can often be seen as a *double sequencing problem.*

In the following subsections, we briefly discuss the distinguishing features of various models found in the literature.

2.2.2 Processing requirements

In very general form, the processing requirements of part j on machine M_i can be modelled by means of a *processing window* $[L_i^j, U_i^j]$. The interpretation is that the time spent by part j on machine M_i must be at least L_i^j, and may not exceed U_i^j. Such processing requirements arise for instance quite naturally in manufacturing processes observed in the electronics industry. Here, printed circuit boards must be sucessively dipped into several chemical baths, and the duration of each chemical bath may neither be too short nor be too long (see e.g. Philips & Unger [1976], Lei [1991]).

Processing windows are general enough to contain many other specifications of the processing requirements as special cases. For instance, the classical situation in which each part has a precisely defined processing time on each machine, and can wait on a machine indefinitely long after it has been processed, arises by setting all upperbounds U_i^j to $+\infty$. Such problems are investigated in several papers; see e.g. Sethi et al. [1992], Hall et al. [1995ab], Sriskandarajah et al. [1995], Crama & Van de Klundert [1995] and the references therein. Also, several authors consider a *no-wait* version of our basic problem, in which all parts are to be unloaded as soon as they finish processing. This can be modelled by setting $L_i^j = U_i^j$; see e.g. Levner, Kats & Meyzin [1995].

In the same spirit as processing windows, production characteristics may also impose restrictions on the material handling activities. In certain environments, the robot may have to load each part on M_i as soon as possible after unloading it from M_{i-1}. Such a restriction is referred to as *robot no-wait.* The alternative is, of course, that the robot may pause between unloading a part from M_{i-1} and loading it on M_i. Observe that such pauses provide a convenient way of respecting the processing windows.

2.2.3 Time models for the robot

In order to completely define an instance of a robotic cell scheduling problem, we need to further specify the behavior of the robot. Several authors (see e.g. Hall et al. [1995ab], Sriskandarajah et al. [1995], Kamoun et al. [1993]) consider the duration of the load and unload activities to be machine dependent: their models assume that it takes ϵ_i time to load or unload a part from machine M_i. More general models could be introduced but, to the best of our knowledge, have not been investigated. On the other hand, simpler models in which all load and unload times are equal ($\epsilon_i = \epsilon$, or even, $\epsilon_i = 0$) are rather common.

An important characteristic of the robot is its travel speed. We denote by δ_{ij} the time needed for the robot to travel from machine M_i to machine M_j. The travel times are often assumed to be symmetric ($\delta_{ij} = \delta_{ji}$ for all i, j) and to satisfy the triangle inequality (for $i < j < k, \delta_{ij} + \delta_{jk} \geq \delta_{ik}$). Some authors also consider models in which the travel time depends on whether the robot is carrying a part or not. This distinction can be modelled in several ways, e.g. by means of a constant correction factor.

In a flowshop, the machines are often laid out in line (see again Figures 2.1 and 2.2). When this is the case, the travel times can be assumed to be symmetric and to satisfy the following relation : for $i < j < k, \delta_{ij} + \delta_{jk} = \delta_{ik}$. In the remainder of this sequel we refer to this equality as the *triangle equality*. In case the triangle equality holds, there may again be a correction factor to model the fact that the robot is carrying a part.

2.2.4 Problem size

The general robotic cell scheduling problem, that will be introduced shortly, is hard to solve so that special, more tractable cases of the problem have received much attention in the literature.

First of all, in many practical instances, the number of machines is relatively small. This justifies the interest in cells consisting of only two or three machines. More generally, regarding the number of machines as a constant, rather than as input data, yields interesting algorithmic and theoretical insights.

Similar remarks hold for the number of distinct parts to be processed. Actually, a good deal of research has been devoted to problems in which all parts are identical, i.e. have the same processing requirements. Notice that in this case, the part input sequencing problem vanishes altogether.

More generally, the parts can often be partitioned into a small number (say, T) of *part types*, with the property that all parts of a same type possess exactly the same processing requirements. A *minimal part set* (MPS) is then described by a vector of the form $(n_1, n_2, \ldots, n_T)$ where n_t indicates the number of parts of type t contained in the MPS ($t = 1, 2, \ldots, T$), and the part set J consists of a number of copies of the MPS (possibly, infinitely many copies) to be produced repeatedly. Very little is known about the complexity of the problems arising in this setting when the number of part types is regarded as a constant.

2.2.5 Objective functions

With very few exceptions (Song et al. [1993] Jeng et al. [1993], who choose to optimize the sum of the completion times), only two classes of production performance measures have been considered in the literature. In order to introduce them, let us denote by S_n ($n = 1, 2, \ldots$) the *completion time* of the n-th part processed in a given schedule, that is the time at which the n-th part is unloaded at the output station. Then, one class of problems has as objective to minimize the *makespan* of the part set J, viz. $\max_{n=1,\ldots,|J|} S_n$, where J is supposed to be finite (see e.g. Kise [1991], Kise et al. [1991]). Another class attempts to minimize the *long run cycle time* of the robotic cell, viz. $\limsup_{n\to\infty} \frac{S_n}{n}$.

Notice that the latter objective looks rather unattractive from an operational viewpoint; indeed, it requires to specify a 'long run' robot move sequence and part input sequence, and these sequences could – a priori – be fairly (infinitely !) long. It is therefore customary to restrict the analysis to 'periodic' part input sequences and 'periodic' robot move sequences with 'short' periods, which can be repeated as long as the part set J is not exhausted. Now, what do we mean by 'short periodic' sequences?

For several reasons, to be addressed in more detail in the next section, it is usual to define the input sequence for only *one* minimal part set (MPS), and to assume that the parts are produced according to this same sequence within *each MPS* making up the entire part set J. Thus, the period of the part input sequence is equal to the number of parts contained in the *MPS*.

Similar comments hold for the robot move sequences considered in the literature: the analysis is mostly restricted to (infinite) repetitions of sequences in which each machine is loaded and unloaded once. A robot move sequence in which each machine is loaded and unloaded k times, and which can be repeated indefinitely, is called a *k-unit cycle* (Hall et al.[1995a]). Notice that, in particular, exactly k parts are produced (i.e., are unloaded at the output station) during each k-unit cycle. In subsequent sections, we deal with k-unit cycle in greater depth.

Another interesting issue is how to compute the long run cycle time when both the part input sequence and the robot move sequence are given. This issue becomes easier to tackle when the sequences are periodic. Indeed, when repeating both sequences, one can reasonably hope that the cycle time (S_n/n) converges, due for instance to some periodic pattern in the completion times of successive parts.

Alternatively, one could restrict the analysis to periodic robot schedules beforehand. As a matter of fact, a sizeable part of the research literature investigates *1-periodic schedules* only, that is schedules in which the time elapsed between consecutive occurences of a same event (e.g., unload machine M_i) is constant over time and independent of the event. The extent to which this restriction leads to suboptimal cycle times will be discussed in the next sections. To the best of our knowledge (and somewhat surprisingly maybe), Song et al. [1993] seem to be the only authors who do not restrict the analysis to 1-periodic schedules beforehand.

2.2.6 Related problems

Not all of the literature dealing with robotic cell scheduling problems is devoted to the bufferless robotic flowshop discussed above. Several interesting papers deal with closely related cells and the corresponding scheduling problems.

First, as already mentioned in the Introduction, there may be more than one robot in the cell. This motivates the extension of the above models to models involving two, three or a larger (constant) number of robots (see Lei & Wang [1991]). When dealing with design problems, the number of robots can even be treated as a decision variable (see Armstrong et al. [1995], Lei et al. [1993], Kats & Levner [1996]).

Several authors study robotic cells in which the machines are equipped with (input or output) buffers where the parts can wait until the machine or the robot becomes available. For instance, Park [1995] uses simulation to investigate the effectiveness of various dispatching rules in such an environment. King et al. [1993] propose a branch & bround algorithm for a similar environment. On the other hand, theoretical work has also been reported on questions that parallel those encountered in the previous subsections. As a general rule, the additional freedom introduced by finite capacity buffers tends to complicate scheduling problems (see e.g. Papadimitriou & Kanellakis [1980]). Buffers may allow, for example, to consider non-permutation schedules, i.e. schedules in which parts are processed in different orders on different machines.

Finally, non flowshop variants of robotic cell problems have also been proposed. For instance, re-entrant flowshops have been studied by Levner [1995] and cells with parallel machines have been considered in Hall et al. [1995c], Jeng et al [1993]. Hertz et al.[1992], consider a re-entrant flowshop in which some of the machines may handle more than one part at a time, and the time elapsed between the end of an activity and the beginning of another one may be bounded. Actually, a robot could theoretically be added to any shop layout, thus giving rise to further interesting models. In addition, the robot can be restricted to travel only along some predefined track (Blazewicz et al. [1992]).

In order to better appreciate the practical relevance of the various robotic cell models and the associated scheduling problems, a survey of case studies in an industrial framework would be of much help, but currently appears to be lacking.

2.3 Optimization problems and models

In the remainder of this chapter, unless explicitly stated otherwise, we only consider no-buffer robotic flowshops. In the current section, we mainly discuss the complexity of, and algorithms for the corresponding scheduling problems when either the part input sequence or the robot move sequence is *not* fixed. To place the results in perspective, we start with a review of the complexity of traditional bufferless flowshop scheduling problems, before proceeding with robotic flowshops.

The best known result in bufferless flowshop scheduling is probably the algorithm proposed by Gilmore and Gomory [1964] for makespan minimization in the two-machine

case. These authors showed that the two-machine problem can be modeled as a special kind of traveling salesman problem (TSP), and proved that such TSPs can be solved in polynomial time. By contrast, the no-buffer three-machine flowshop problem is strongly NP-complete. As observed by McCormick et al. [1993], this follows from a result of Papadimitriou and Kanellakis [1980], who proved that two-machine flowshop scheduling with a unit capacity buffer is strongly NP-complete.

The situation is quite similar for no-wait flowshop scheduling problems. In the two-machine case, there is no difference between no-wait and no-buffer, and for three machines the problem is strongly NP-complete (Rock [1984]). For any number of machines, no-wait flowshop scheduling problems can be modeled as TSPs. More details can be found in a paper by Kamoun and Sriskandarajah [1993], who discuss a variety of no-wait and no-buffer scheduling problems.

Finally, although this situation has not received much attention in the literature, let us consider the case where the processing requirements of a traditional no-buffer flowshop are expressed by means of processing windows. In the two-machine case, the problem can be solved by the Gilmore-Gomory algorithm, since there is always a no-wait schedule among the optimal ones. On the other hand, since it contains the no-wait problem as a special case, the problem with processing windows is strongly NP-complete for three or more machines.

All the complexity results mentioned above concern the makespan minimization objective. McCormick et al. [1993] address the relation between makespan minimization and cycle time minimization for traditional flowshop scheduling problems. They establish that makespan minimization problems can be efficiently transformed to cycle time minimization problems. Thus, the latter model is more general than the former one. Their result however, implicitly restricts the analysis to 1-unit part input sequences. In the sequel, we adopt the latter restriction since this is common practice. Moreover, tacitly assuming that the complexity of makespan and cycle time minimization problems are of the same order, we only consider cycle time minimization problems in the following. Our assumption could lead to erroneous conclusions in the unlikely event that the cycle time minimization objective makes the problem substantially harder than makespan optimization does.

As a final remark on ordinary flowshop scheduling problems, let us briefly consider problems with few parts or few part types. When there are only few parts (i.e., when the number of parts is bounded from above by a constant), complete enumeration of all part input sequences yields a polynomial solution strategy for the problem. On the other hand, when the number of part types (as opposed to the number of parts) is fixed, then the total number of part input sequences can be exponential in the length of the encoding of an instance, and brute force enumeration becomes inefficient. Therefore, such problems (sometimes referred to as high multiplicity problems) have their own interesting characteristics from the viewpoint of algorithmic complexity; see e.g. Agnetis et al. [1989] and Van de Klundert [1995] for a more detailed treatment.

By contrast with the above discussion, we shall see that, in robotic cells, nontrivial problems arise even when the number of parts in the minimal part set is small. Actually,

even when there is a *unique* part type (a vacuous situation in traditional flowshops), the question remains of specifying an optimal (or feasible) robot move sequence. At this point, for a good understanding of the problem, it becomes useful to discuss robot move sequences in some more depth before we proceed.

Remember that the robot performs three kinds of operations: loading, unloading and transportation of parts. Since the robot can only handle one part at a time, the load/unload operation that directly precedes the loading of machine M_{i+1}, is necessarily the unloading of machine M_i, $i = 0, \ldots, m$.

Definition 2.1 The sequence of robot moves

1. unload M_i ,
2. travel from M_i to M_{i+1},
3. load M_{i+1}

is called *(robot) activity* A_i, for $i = 0, \ldots, m$,

Definition 2.2 An infinite sequence π of activities $A_0, \ldots, A_m$ is called a *feasible robot move sequence* if,

1. the robot never has to unload any empty machine,
2. the robot never has to load any loaded machine.

Notice that infinite sequences may be implicitly defined by giving a shorter sequence that is to be repeated infinitely many times. Since a 1-unit cycle is a feasible robot move sequence in which each machine is loaded and unloaded exactly once, we can view every (feasible) 1-unit cycle as a permutation of the activities $A_0, A_1, \ldots, A_m$. Interestingly, the converse statement is also true, i.e.

Theorem 2.1 (Lieberman & Turksen [1981], Sethi et al. [1992]) Every permutation of the activities $A_0, A_1, \ldots, A_m$ is a 1-unit cycle.

Proof. We must show that every permutation of the activities satisfies the two conditions of Definition 2.2, that it loads an unloads each machine exactly once, and that it can be repeated infinitely many times. By the definition of an activity, each machine is loaded and unloaded exactly once in every permutation of the activities. Now, consider some permutation of the activities, say π. It remains to show that π can be indefinitely repeated, without unloading unloaded machines or loading loaded machines. Without loss of generality, assume that A_0 is the first activity in π, and assume that, when the robot starts executing π, the only machines that contain a part are those machines M_i such that A_i precedes A_{i-1} in π (these are the machines that must be unloaded before they are loaded). A moment of reflection then shows that, in the course of executing π, the robot only unloads machines that actually contain a part. Moreover, it can be seen

that when the robot has finished executing π, then the flowshop has returned to its original state (since each machine has been loaded and unloaded exactly once). Thus, we conlude that π can be repeated infinitely many times, without unloading unloaded machines or loading loaded ones, provided that the initial loaded/unloaded state of the machines is as defined above. ■

In the remainder we always assume that we are free to specify the initial loaded/unloaded state of the machines. This yields specifically that we do not have to assume that the cell is initially empty.

The most general robotic cell scheduling problem we study in this chapter is as follows. The robot as well as the machines can contain only one part at a time, and there are no buffers in the cell. We assume that travel distances for the robot between machines are given by means of a symmetric *distance matrix* D, whose elements δ_{ij} satisfy the triangle equality. Further, there is a *loading and unloading time* ϵ_i for each machine $M_i, i = 0, \ldots, m+1$, where m is the number of machines in the cell. The processing requirements of part j, $j = 1, \ldots, n$ on machine $M_i, i = 1, \ldots, m$ are given by means of processing windows $[L_i^j, U_i^j]$. Further we assume in this most general version that every feasible robot move sequence and every part input sequence is allowed. Although, of course, parts have to produced in the ratio implied by the minimal part set. The objective is to minimize the average long run cycle time. More formally the optimization version can be stated as follows :

Definition 2.3 *General Robotic Flowshop Scheduling Problem* (GRFS):

INPUT : $D, \epsilon_i, [L_i^j, U_i^j]$ $(i = 0, \ldots, m+1,\ j = 1, \ldots, n)$

QUESTION : Find a feasible robot move sequence and part input sequence with minimum long run cycle time.

The adjective general is intended to reflect that this is the most general problem we consider. The definition may be short and general, it is far from convenient. For instance, it is (at this point) not at all clear how to determine the long run cycle time when the inputs and both the sequences are given. If we add the fact that specifying long sequences is not desirable from a practical viewpoint, it is no surprise that many authors have considered simpler models.

A restriction that is common to all research known to the author is that the part input sequence is restricted to be an n-unit cycle : the order in which the parts constituting an MPS are taken from the input device is the same for every MPS (and hence repeats after every n parts). A second often imposed restriction is that the robot move sequence consists of indefinitely repeating a 1-unit cycle. This implies in particular that the order of the robot acticivities does not vary with the part being taken from the input device. There are only very few cases in which this restriction is known not to cause an increase in the minimum possible long run cycle time. In many cases

the objective of minimizing long run cycle time is still troublesome even when making both restrictions mentioned above. Therefore many authors do not only restrict the sequences to be a repetition of shorter sequences, but also impose some repetitive pattern in the time intervals elapsing between consecutive executions of identical load/unload operations. In view of these considerations, the introduction of schedules will turn out to be a convenient way of dealing with long run cycle time.

Definition 2.4 A *schedule* S is defined as a specification of starting times for each load and unload operation. More specifically, we denote by $S(l,i,t)$ $(S(u,i,t))$ the time at which the t-th loading (unloading) of machine M_i starts in schedule S $(i = 0,\ldots,m,\ t \in \mathbb{N})$.

Let us study conditions under which a schedule S is feasible. The following conditions ensure that between loading and unloading of a machine, the machine has enough time for processing. Let part j be the part loaded onto machine M_i in the t-th loading operation. If the t-th loading takes place before the t-th unloading, then :

$$S(u,i,t) - S(l,i,t) - \epsilon_i \geq L_i^j, \tag{2.1}$$

and

$$S(u,i,t) - S(l,i,t) - \epsilon_i \leq U_i^j. \tag{2.2}$$

If the t-th unloading takes place before the t-th loading, then :

$$S(u,i,t+1) - S(l,i,t) - \epsilon_i \geq L_i^j, \tag{2.3}$$

and

$$S(u,i,t+1) - S(l,i,t) - \epsilon_i \leq U_i^j. \tag{2.4}$$

Furthermore, for a feasible schedule S, let $\pi(S) = \pi$ be the *unique* feasible robot move sequence in which the machines are loaded in the same order as in S. (In a feasible schedule loading operations do not take place simultaneously.) We say that S is a *schedule for* π. The schedule S for π must allow the robot enough travel time between consecutive load/unload operations :

$$S(u,i,t) + \epsilon_i + \delta_{i,i+1} \leq S(l,i+1,t). \tag{2.5}$$

Furthermore, if the t-th execution of A_k (directly) succeeds the s-th execution of A_i in π then

$$S(l,i+1,s) + \epsilon_{i+1} + \delta_{i+1,k} \leq S(u,k,t). \tag{2.6}$$

If $\pi(S)$ is a feasible robot move sequence, conditions (2.1)-(2.6) are necessary and sufficient for feasibility of S : the time intervals elapsing between loading and unloading do not violate the processing windows because of (2.1)-(2.4), and the robot is allowed enough time for travelling between load and unload operations because of (2.5)-(2.6). Moreover, since π is a feasible robot move sequence, the robot does not unload any unloaded machines, nor does it load any loaded machines.

Definition 2.5 A schedule S is said to be *r-periodic* if there exists some constant C_S, called the *cycle time* of S, such that $S(l,i,t+r) - S(l,i,t) = r \times C_S$ and $S(u,i,t+r) - S(u,i,t) = r \times C_S$ for all $0 < i < m+1, t \in \mathbb{N}$.

Obviously, for an r-periodic schedule S the long run cycle time equals C_S. (Hall et al.[1995ab] use steady state for what we call 1-periodic.)

A third often made restriction is now to only consider n-periodic sequences (recall that n is the number of parts in the MPS). This restriction will turn out to be particularly convenient in the special case of identical parts, a fourth often imposed restriction. Notice that in such cases n-periodic reduces to 1-periodic. Special cases of *GRFS* in which all these four restrictions are imposed have received considerable attention in the literature. In the following sections we keep track of whether relaxing the third restriction does or does not yield better solutions.

2.3.1 Processing windows and identical parts

In this section we discuss the complexity of the special case of *GRFS* in which there is only one part type. Thus, since all parts have identical processing requirements, we denote the processing windows as $[L_i, U_i]$. This problem is studied in a series of papers which is to be addressed shortly. In this subsection we give an NP-completeness proof for a rather general version of this problem.

As customary, we restrict the analysis to 1-periodic schedules, and consequently to 1-unit cycles. If we let $\pi(S)$ be the 1-unit cycle induced by the 1-periodic schedule S (more precisely, we denote now by $\pi(S)$ the unique 1-unit cycle which, when repeated infinitely many times, defines the same sequences of load/unload operations as the sequence induced by S), conditions (2.1) – (2.6) translate as follows :

If A_{i-1} precedes A_i in $\pi(S)$, then

$$S(u,i,t) - S(l,i,t) - \epsilon_i \geq L_i, \tag{2.7}$$

$$S(u,i,t) - S(l,i,t) - \epsilon_i \leq U_i. \tag{2.8}$$

On the other hand, if A_i precedes A_{i-1} in $\pi(S)$, then

$$S(u,i,t) + C_S - S(l,i,t) - \epsilon_i \geq L_i, \tag{2.9}$$

$$S(u,i,t) + C_S - S(l,i,t) - \epsilon_i \leq U_i. \tag{2.10}$$

The robot must again be allowed enough time to perform each activity :

$$S(u,i,t) + \epsilon_i + \delta_{i,i+1} \leq S(l,i+1,t). \tag{2.11}$$

Furthermore, if A_k is the activity succeeding A_j in $\pi(S)$ then

$$S(l,j+1,t) + \epsilon_{j+1} + \delta_{j+1,k} \leq S(u,k,t), \tag{2.12}$$

and, if A_j is the last activity in $\pi(S)$, and A_k the first,

$$S(l, j+1, t) + \epsilon_{j+1} + \delta_{j+1,0} \leq S(u, k, t) + C_S. \tag{2.13}$$

Notice that (2.13) arises from (2.6), since $\pi(S)$ is repeated infinitely many times.

Thus, in the special case under consideration, the goal is to find a schedule S, that minimizes C_S subject to (2.7) – (2.13). Notice that, since this is a linear programming problem, the optimal cycle time can be computed in polynomial time once π is known. Hence, for each π, let C_π be the minimum long run cycle time attainable by a schedule satisfying (2.7) – (2.13) in which the activities are performed in the order dictated by π. Then, the problem under consideration can be defined as :

Definition 2.6 *Robotic flowshop scheduling problem for identical parts (RFSI)* :

INPUT : ϵ_i, L_i, U_i, D, constant C.

QUESTION : Is there a permutation π of the activities such that $C_\pi \leq C$?

Let us briefly review the literature in which this problem and closely related ones are addressed. The problem was introduced by Philips and Unger [1976]. They formulate the problem as an integer linear program, and solve some instances using standard software. Lieberman & Turksen [1981] formulate several related problems, e.g. problems in which there is more than one robot or problems in which the cell is not restricted to be a flowshop. Song et al. [1993] propose heuristics to find the optimal k-unit feasible robot move sequence for the no-wait version of this problem. In Lei [1993], the problem of minimizing the cycle time for a given permutation of the activities is shown to be solvable in $O(m^2 \log m \log B)$, where B depends linearly on the input parameters. Lei and Wang [1994], Armstrong, Lei and Gu [1994] and Hanen and Munier [1994] discuss branch & bound procedures for RFSI and alike. In Lei, Armstrong and Gu [1993], and in Lei and Wang [1991], heuristic procedures for a similar problem with multiple robots are given.

As promised, we show in this section that RFSI is NP-Complete. An alternative proof can be found in Lei and Wang [1989]. However, the instances created by the reduction in Lei and Wang [1989] are somewhat artificial : there may be non zero robot travel time between consecutive activities A_i (which ends with loading M_{i+1}) and activity A_{i+1} (which starts with unload machine M_{i+1}).

Theorem 2.2 RFSI is strongly NP-Complete.

Proof. Membership in NP follows from (2.7) – (2.13) (see e.g. Lei [1993]). We show its completeness by giving a reduction from the Bin Packing Problem to RFSI.

Bin Packing :

INPUT : Finite set $V = \{v_1, \ldots, v_q\}$ of items, a size $s(v) \in \mathbb{Z}^+$ for each $v_i, i = 1, \ldots, q$, positive integer $K \leq q$ and a positive integer B.

QUESTION : Is there a partition of V into disjoint sets $V_1, \ldots, V_K$ such that the sum of the sizes of the items in each V_i is B or less?

Construct an instance of RFSI as follows. There are $m = 2q+1+2K$ machines. The processing windows of the machines $M_1, \ldots, M_K$ and machines $M_{2q+K+2}, \ldots, M_{2q+2K}$ are $[(4q + 2K + 3)B, (4q + 2K + 3)B]$. The processing window of $M_{2q+2K+1}$ is $[(4q + 2K + 2)B, (4q + 2K + 2)B]$. The windows of the machines $M_{K+2i+1}, i = 0, \ldots, q$ are $[0, +\infty]$. Finally the windows of the machines $M_{K+2i}, i = 1, \ldots, q$ are $[s(v_i), s(v_i)]$. The loading and unloading times ϵ_i are all equal to 0. The travel time between two adjacent machines equals B.

The following claim will be useful in the remainder of the proof :

Claim 2.1 For all $0 \leq i < K$, if A_i precedes A_{i+1} then activities $A_j, j \geq 2q+K+i+2$ cannot be performed between the execution of A_i and A_{i+1}.

Proof. The processing window of machine M_{i+1} is $[(4q + 2K + 3)B, (4q + 2K + 3)B]$, and hence $(4q + 2K + 3)B$ time units after loading the machine it must be unloaded. To perform an activity with index at least $2q + K + i + 2$ the robot must travel to machines with index at least $2q + K + i + 3$. Travelling to machine with index at least $2q + K + i + 3$ and back *between* the execution of A_i and A_{i+1}, requires at least $2 \times (2q + K + 2)B > (4q + 2K + 3)B$ time, causing the schedule to be infeasible. ∎

Claim 2.2 There is a 1-periodic schedule with cycle time $(4q + 2K + 4)KB + (4q + 3K + 4)B$ if and only if the bin packing instance is a yes instance.

Proof. To prove the claim, we first show that the activities

$$A_0, \ldots, A_K, A_{2q+K+2}, \ldots, A_{2q+2K+1}$$

must be in some specific order in every permutation that denotes a solution with the desired cycle time. Without loss of generality we may assume A_0 to be the first activity in the permutation. We claim activities A_0 to A_K are in order of their index in every feasible solution. Suppose not: let $i \in \{1, \ldots, K-1\}$ be the smallest index for which A_{i+1} precedes A_i (notice that $i \neq 0$). We consider two cases :

1. $A_{2q+2K+1}$ is scheduled before A_i. Let $A_j, j \leq i$ be the activity such that $A_{2q+2K+1}$ takes place between A_{j-1} and A_j. It follows from the Claim 2.1 that the schedule is infeasible.

2. $A_{2q+2K+1}$ is scheduled after A_i. This implies that between the execution of A_i in some iteration of the schedule and the execution of A_{i+1} in the next execution of the schedule the robot must perform $A_{2q+2K+1}$ and A_0 in that order. It follows again that the total travel time between A_i and A_{i+1} causes the schedule to be infeasible.

Thus activities $A_0, \ldots, A_K$ must indeed be in increasing order of their index. It is also straightforward to check that $A_{2q+K+1}, A_{2q+K+2}, \ldots, A_{2q+2K+1}$ must occur in this order in any feasible schedule (if $A_{2q+K+i+1}$ is performed before A_{2q+K+i}, then the travel time from the machine $M_{2q+K+i+1}$ to M_0 and back exceeds the processing window of $M_{2q+2K+i+1}$).

Let A_0 start at time 0. Considering the processing windows of machines $M_1, \ldots, M_K$ we can derive that the robot cannot start performing activity $A_i, i = 1, \ldots, K$ before time $(4q + 2K + 3)iB + iB$. More specifically, A_K cannot be started before $(4q + 2K + 3)KB + KB$. It can also be concluded from Claim 2.1 that $A_{2q+2K+1}$ cannot take place before A_K since otherwise the schedule would be infeasible. Combining these two observations leads to the conclusion that the total cycle time must be at least $(4q + 2K + 3)KB + KB + (2q + K + 1)B + B + (2q + 2K + 2)B = (4q + 2K + 4)KB + (4q + 3K + 4)B$. Thus we have proved that this quantity (see Claim 2.2) is a lowerbound on the cycle time.

Claim 2.1 implies that $A_{2q+K+1+i}$ cannot precede A_i in any solution, for $i = 1, \ldots, K$. We are now going to show that A_{2q+2K} must precede A_K in every schedule having the desired cycle time. First of all, observe that A_{2q+K+1} cannot precede A_K. Hence if A_{2q+2K} is scheduled after A_K, it is either scheduled between A_K and $A_{2q+2K+1}$ or after $A_{2q+2K+1}$. In the latter case, the schedule was shown above to be infeasible. Thus A_{2q+2K} is scheduled between A_K and $A_{2q+2K+1}$. Now the total cycle time is at least the sum of the following time periods :

1. The interval from A_0 to the start of A_K, performance of A_K and travel time to machine M_{2q+2K} : taking time $(4q + 2K + 4)KB + B + (2q + 2K - (K))B$,

2. perform A_{2q+2K}, and wait or do something else until machine $M_{2q+2K+1}$ has finished processing : $B + (4q + 2K + 2)B$,

3. Unload $M_{2q+2K+1}$, bring the part to the ouput device and travel back to the input device to start the next execution of A_0 : $B + (2q + 2K + 1)B$.

This would result in a total cycle time of at least $(4q + 2K + 4)KB + B + (2q + 2K - K)B + B + (4q + 2K + 2)B + B + (2q + 2K + 1)B > (4q + 2K + 4)KB + (4q + 3K + 4)B$. Now, since Claim 2.1 implies that A_{2q+2K} cannot precede A_{K-1}, we know that activities A_{K-1}, A_K and A_{2q+2K} are in the order A_{K-1}, A_{2q+2K}, A_K. It is easy to check that $A_{2q+2K-1}$ cannot be scheduled between A_{K-1} and A_K too. Moreover, since $A_{2q+2K-1}$ cannot be scheduled before A_{K-2}, as results from Claim 2.1, and cannot be scheduled after A_{2q+2K}, it must be scheduled between A_{K-2} and A_{K-1}. An inductive argument then establishes that $A_{2q+2K-i}$ takes place between A_{K-i-1} A_{K-i}. Hence

we conclude that in any schedule that achieves the desired cycle time, the activities $A_0, \ldots, A_K, A_{2q+2K+1}, \ldots, A_{2q+2K+1}$ must be performed in the order

$$A_0, A_{2q+K+1}, A_1, A_{2q+K+2}, \ldots, A_K, A_{2q+2K+1}.$$

The remainder of the proof is now to show how the other activities must be plugged in, so that a schedule with the desired cycle time is obtained if one exists, and that such a schedule exists if and only if the bin packing instance is a yes instance.

We make three observations :

1. A_{K+2i-1} and $A_{K+2i}, i = 1, \ldots, q$ must always occur consecutively in every feasible permutation, since $s(v_i) \leq B$.

2. All the trajectories traveled between the end of A_i and the start of $A_{i+1}, i = 0, \ldots, K-1$, can be traveled only twice, since otherwise the processing window of M_{i+1} is violated. This implies that we cannot schedule any activities between A_{2q+K+i} and A_i for $i = 1, \ldots, K$.

3. None of the activities A_i with index $K+1 \leq i \leq K+2q$ can be scheduled after $A_{2q+2K+1}$ since otherwise the cycle time will be too large.

Together, these three observations imply that we must schedule $K+1$ sets of pairs of consecutive activities between A_i and $A_{K+2q+i+1}$, for $i = 0, \ldots, K$. The total travel time between loading M_{i+1} in A_i and unloading M_{i+1} in $A_{i+1}, i = 0, \ldots, K-1$ amounts $(4q+2K+2)B$. In view of the processing windows, this leaves us B time to perform pairs of activities A_{K+2i+1}, A_{K+2i+2}. Between any such pair of activities the robot must wait $s(v_{i+1})$ time. Furthermore, we cannot schedule any activities after the execution of A_K, because of the processing window of $M_{2q+2K+1}$. Thus, a schedule with the desired cycle time exists if and only if the numbers $s(v_i)$ can be partitioned into K sets each having weight no more than B. ∎

Remark 2.1 In fact we have proved more than we promised. In the 'yes' instances created through the reduction 1-periodic schedules have minimum long run cycle time over all feasible schedules for 1-unit cycles, periodic or not. Thus we may conclude that finding a feasible schedule, 1-periodic or not, is strongly NP-complete.

2.3.2 No-wait time windows and identical parts

As before, if all parts are identical, the time windows can be represented by $[L_i, U_i], i = 1, \ldots, m$. In this subsection we briefly review the literature on the special case of GRFS where all parts are identical and $U_i = L_i$ for $i = 1, \ldots, m$. Levner et al. [1996] present an algorithm that solves this problem in $O(m^3)$ time. Their approach is to identify a set of 'forbidden intervals' for the cycle time. Based on these intervals they are able to identify a smallest attainable cycle time, and a 1-unit cycle that achieves this cycle time. These authors have extended their results in several directions. They propose

strongly polynomial algorithms for extensions to re-entrant flowshops and problems with more than one robot. Moreover, Kats & Levner [1996] propose a solution method to simultaneously optimize over the number of robots in the flowshop and the cycle time in $O(m^5)$ time.

2.3.3 Unbounded time windows

In this subsection we consider the special case of GRFS where $U_i^j = +\infty$ for all i, j. This special case, in which the machines behave as is customary in classical scheduling problems, has given rise to a very fruitful research area. In the practical environments that initiated this research, the machines are placed in a line or a circle, and therefore, the distances are often assumed to satisfy the triangle equality. Throughout this subsection, we shall again assume it to hold. In a seminal paper, Sethi et al.[1992] introduce the problem and study special cases in which there are only two or three machines. They show that the problem of finding an optimal part input sequence, for a given 1-unit cycle can be solved in the case where there are only two machines. Their algorithm is an extension of the Gilmore and Gomory algorithm for the ordinary two machine no-buffer flowshop (Gilmore and Gomory [1964]). They also explain how the problem can be solved by enumeration when there is only one part type and three machines. Moreover, they derive that, in this case, two of the six possible 1-unit cycles never are uniquely optimal. These results have been extended in two directions.

As shown in Chapter 3, the results for identical parts can be generalized. Indeed, in a robotic cell with m machines, there exists a set of 1-unit cycles of cardinality 2^{m-1} that necessarily contains an optimal solution. This is the set of so-called 'pyramidal permutations' of the activities. In Chapter 3, we also give algorithms to find an optimal 1-unit cycle for a cell with m machine in $O(m^3)$ or $O(m^2 \log(dm))$ time, where d depends on D.

Sriskandarajah et al. [1995], Hall et al. [1993,1995a,1995b], Kamoun et al. [1993], and Ioachim & Soumis [1995], have studied at length problems with multiple part types. Hall et al. [1995a] show that in a cell with two machines, it is possible to simultaneously optimize the part input sequence and the 1-unit cycle in polynomial time. Their algorithm uses again the Gilmore and Gomory algorithm, and indeed finds an optimal way to switch between the two possible 1-unit cycles in a two machine cell. These authors also show that the problem is NP-hard in a robotic cell with 3 or more machines. In fact they identify 1-unit cycles in a cell with 3 machines for which it is strongly NP-hard to find an optimal part input sequence. These results are extended to cells with more than three machines. For each possible 1-unit cycle in a robotic cell with m machines, they either give an algorithm to determine the optimal part input sequence in time polynomial in m, or show that it is NP-hard to find the optimal part input sequence.

2.4 Cycle time computation

In this section we study the following question : Given a feasible robot move sequence and a part input sequence, what is the smallest long run cycle time attainable by a feasible schedule? We start again our quest with the most general problem in this setting, which is studied in Lei [1993]. We then study the same problem, and more general ones, in a cell with unbounded time windows.

2.4.1 Time windows and identical parts

The problem of determining the minimum long run cycle time for given robot move and part input sequences is more general than related problems studied in the literature. Often the analysis is restricted to 1-periodic schedules and to problems in which there is only one part type. For instance, we know already that for RFSI conditions (2.7) – (2.13) are necessary and sufficient for feasibility of a 1-periodic schedule S. Hence, the linear program in which we minimize C_S subject to these conditions is an obvious model for the cycle time minimization problem. Lei [1993] proposed an algebraic algorithm to solve the LP formulation in $O(m^2 \log m \log B)$ time, where B depends polynomially on the bounds of the processing windows. Levner [1995] proposes a PERT algorithm to solve the LP, which is based on an algorithm of Levner and Nemirovsky [1994]. Of course, these results also hold for the special case in which $L_i^j = U_i^j$, no-wait time windows, or $U_i^j = +\infty$, unbounded time windows. For the latter case however, we overview the literature and present a different algorithm in the next subsection.

2.4.2 Unbounded time windows

In the case where the upper bounds on the processing windows are plus infinity, Sethi et al. [1992] present an approach based on minimax algebra for cycle time minimization. Cunninghame-Green [1979] provides an excellent reference on minimax algebra, which is a powerful tool to analyse scheduling and other problems. Cohen et al. [1985] use this tool to study the behavior of cyclic schedules in a manufacturing system with pallets, and they show how to use an algorithm of Karp [1978] (for finding a minimum weight digraph cycle) in order to actually compute the smallest feasible cycle time.

Apart from showing how to determine an optimal part input sequence for various fixed robot sequences, Hall et al. [1993] also give several algorithms to compute the cycle time for given 1-unit cycles in a three machine cell. Ioachim & Soumis [1995] propose a different approach to compute cycle time, based on a PERT-like model. The algorithm they propose is able to handle problems with an arbitrary number (n) of part types and an arbitrary number (m) of machines. The running time of this algorithm is $O(n^3m^3)$ in case the robot is restricted to repeating 1-unit cycles. All the above authors restrict the analysis to periodic schedules.

The remainder of this subsection is devoted to yet another model, foreshadowed in Crama & Van de Klundert [1995], which is only slightly different from the model

of Ioachim & Soumis [1995]. Based on this model we establish in this section that, if the robot move sequence repeats an n-unit cycle, there are n-periodic schedules that achieve the minimum long run cycle time. Moreover, we give an algorithm to compute an n-periodic schedule with mimimum cycle time. This algorithm dominates all the previous ones in terms of computational complexity. It relies on the above mentioned algorithm of Karp [1978] to find a maximum weight digraph cycle (where the weight of a cycle is its length divided by the number of arcs it consists of). The results we obtain in this section may also be obtained through max algebra, but we believe our model to be more transparent.

In the remainder of this section we denote by π the given n-unit cycle. Given π, we construct a directed graph $G(V, A)$ as follows (see Figure 2.3). We associate a vertex v_i^j with (the beginning of) activity A_i^j, where A_i^j denotes the j-th execution of activity A_i in the n-unit cycle, for all $i = 0, \ldots, m, j = 1, \ldots, n$. There is an arc from vertex v_i^j to vertex v_l^k if activity A_l^k directly succeeds A_i^j in the n-unit cycle, or if $k = 1$, $j = n$, A_l^1 is the first activity in the sequence and A_i^n is the last activity in the sequence. These arcs form a Hamiltonian cycle in G (see Figure 2.3). We use them to model inequalities (2.5) and (2.6) as follows. The length of arc (v_i^j, v_l^k) is set equal to $\epsilon_i + \delta_{i,i+1} + \epsilon_{i+1} + \delta_{i+1,l}$: this respects the time the robot takes to perform A_i (see restriction (2.5)), and subsequently travel to machine M_l to perform A_l (see restriction (2.6)).

Restrictions (2.1)-(2.5) are modelled by another set of arcs, as follows. Consider an arbitrary part p ($p = 1, \ldots, n$) which is unloaded from M_0 in the first execution of π. For each $i = 0, \ldots, m-1$ there is a $j \in \{1, \ldots, n\}$ such that this part is loaded onto machine M_{i+1} during the $j + ln$-th execution of A_i, for some $l \in \mathbb{N}$ (Observe that part p could be loaded onto machine M_{i+1} in the second repetition of π, which would give rise to the case $l = 1$, The same part will be subsequently unloaded from M_{i+1} in the $k + qn$-th execution of A_{i+1}, for some $k = 1, \ldots, n$ $q \in \mathbb{N}$. We then draw an arc (v_i^j, v_{i+1}^k) with length $\epsilon_i + \delta_{i,i+1} + \epsilon_{i+1} + L_{i+1}^p$ (where L_{i+1}^p is, as usual, the processing time of part p on machine M_{i+1}). The term $\epsilon_i + \delta_{i,i+1}$ models again condition (2.5), while $\epsilon_{i+1} + L_{i+1}^p$ models conditions (2.1) and (2.3) (notice that conditions (2.2) and (2.4) are vacuous for infinite upper bounds). This process assigns a part in the part input sequence to each v_i^j, and an outgoing arc of appropriate length to model inequalities (2.1)-(2.5).

Observe that there are two arcs from v_i^j to v_{i+1}^k if the two corresponding activities concern the same part. In such a case the shortest of these two arcs (i.e. the robot travel time arc) is deleted.

After this modification G is a directed cyclic graph. In fact, G is a cyclic PERT graph, which completely models the necessary and sufficient conditions given in the previous section, under the assumption of infinite upper bounds on the processing windows. Informally we notice that, indeed, the length of a path from v_i^j to v_l^k in G is a lowerbound on the time that the j-th execution of A_i and the k-th execution of A_l must be apart. In order to make this statement more precise, let us first adapt slightly the definition of 'schedule' that we used in the previous section (cf. Definition 2.4).

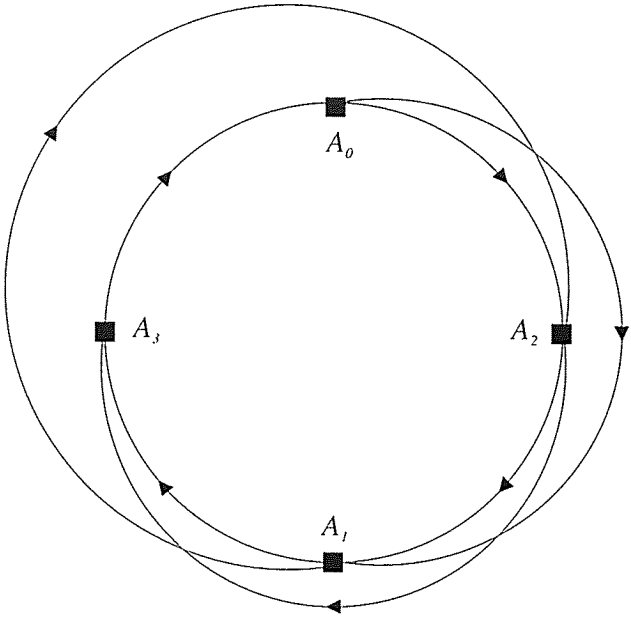

Figure 2.3: The graph associated with A_0, A_2, A_1, A_3

Definition 2.7 A schedule for an n-unit cycle is a real valued function S on $\{(A_i^j, t) | i = 0, \ldots, m, j = 1, \ldots, n, t \in \mathbb{N}\}$, where $S(A_i^j, t)$ denotes the starting time of A_i^j in the t-th repetition of the n-unit cycle.

(Observe that in the present framework with infinite upperbounds on all processing windows, we can freely assume that each part is loaded on machine M_{i+1} exactly $(\epsilon_i + \delta_{i,i+1})$ time units after it has been unloaded from machine M_i. Hence, there is no need to keep track of loading start times, as was previously the case.)

Consider now any directed path P, with origin v_i^j in G (possibly with repeated arcs and/or vertices). We say that P goes around G t times if P contains exactly t arcs of the form (v_l^k, v_s^r), where $A_l^k \neq A_i^j$ and, in π:

1. A_l^k precedes A_i^j and A_i^j precedes A_s^r, or
2. A_i^j precedes A_s^r and A_s^r precedes A_l^k, or
3. A_s^r precedes A_l^k and A_l^k precedes A_i^j.

(Intuitively, such a path wraps around the Hamiltonian cycle of G, and goes t times over vertex v_i^j.)

Denote by $L(v, u, t)$ the length of the longest path from v to u, $(u, v \in V)$ that goes around G t times, $t \in \mathbb{N}$. It is not hard to see that :

Lemma 2.1 Let S be any schedule on $\{(A_i^j, t) | i = 0, \ldots, m,\ j = 1, \ldots, n,\ t \in \mathbb{N}\}$. Then S is a feasible schedule for π if and only if

$$L(v_i^j, v_l^k, t) \leq S(A_l^k, t) - S(A_i^j, 0)$$

whenever the j-th execution of A_i precedes the k-th execution of A_l in π, $t \in \mathbb{N}$ and

$$L(v_i^j, v_l^k, t-1) \leq S(A_l^k, t) - S(A_i^j, 0),$$

whenever the k-th execution of A_l precedes the j-th execution of A_i in π, $t \in \mathbb{N}$.

Proof. Trivial.

In particular, if we divide by n the length of a path from v_i^j to v_i^j that goes around G once, we obtain a lowerbound on the cycle time of the n-unit cycle. More generally, denote by C the minimum long run cycle time of any feasible schedule S for π. Then:

$$n \times C \geq \max_{t \in \mathbb{N}} \max_{v \in V} \frac{L(v, v, t)}{t} \stackrel{\text{def}}{=} L^*. \tag{2.14}$$

Now, any path that goes around G at least once must contain an arc of type (v_i^n, v_l^1). There are $m+1$ vertices of the form v_l^1, one for each $l = 0, \ldots, m$. Thus, each cycle that goes around G more than $m+1$ times must contain one of these vertices twice, by

the pigeonhole principle, and hence can be split into several simple cycles. Therefore, we have:

$$L^* = \max_{1 \le t \le m+1} \max_{v \in V} \frac{L(v,v,t)}{t}. \tag{2.15}$$

Let us first concentrate on the construction of a feasible n-periodic schedule when this maximum is known. Suppose that v^* and t^* realize the maximum in (2.15), i.e.

$$L^* = \frac{L(v^*, v^*, t^*)}{t^*}.$$

For $i = 0, \ldots, m$ and $j = 1, \ldots, n$, we define

$$S(A_i^j, 0) = \max_{0 \le t \le m} \{L(v^*, v_i^j, t) - t \times L^*\}, \tag{2.16}$$

$$S(A_i^j, s) = S(A_i^j, 0) + s \times L^*, \text{ for all } s \in \mathbb{N}. \tag{2.17}$$

Lemma 2.2 S is an n-periodic schedule for π with cycle time $\frac{L^*}{n}$.

Proof. Shift π such that the activity corresponding to v^* is the first activity in π. Let A_i^j and A_l^k be arbitrary activities. We deal here with the case that A_i^j precedes A_l^k in π (the other case being left to the reader). According to Lemma 2.1, we must show in this case that S satisfies

$$L(v_i^j, v_l^k, t) \le S(A_l^k, 0) + t \times L^* - S(A_i^j, 0), \tag{2.18}$$

for all $t \in \mathbb{N}$.

Now, choose $s, 0 \le s \le m$, such that

$$S(A_i^j, 0) = L(v^*, v_i^j, s) - s \times L^*, \tag{2.19}$$

and observe that, for all $t \in \mathbb{N}$

$$L(v^*, v_i^j, s) + L(v_i^j, v_l^k, t) \le L(v^*, v_l^k, t+s). \tag{2.20}$$

Now, we claim that

$$L(v^*, v_l^k, t+s) \le S(A_l^k, 0) + (t+s) \times L^*. \tag{2.21}$$

Indeed, in case $s + t \le m$, this is true by definition of S. In case $s + t > m$ we have again that there exists a vertex w that appears twice on the longest path from v^* to v_l^k. Let t' be the number of times the path goes around G between these two occurrences of w. By definition of L^*, $L(w, w, t') \le t' \times L^*$. Hence, (2.21) follows by induction from

$$L(v^*, v_l^k, t+s-t') \le S(A_l^k, 0) + (t+s-t') \times L^*.$$

Combining (2.20) and (2.21), we derive

$$L(v^*, v_i^j, s) + L(v_i^j, v_l^k, t) \leq L(v^*, v_l^k, t+s) \leq S(A_l^k, 0) + (t+s) \times L^*. \tag{2.22}$$

Hence, from (2.19) and (2.22),

$$L(v_i^j, v_l^k, t) \leq S(A_l^k, 0) + t \times L^* - S(A_i^j, 0),$$

as required. ∎

Theorem 2.3 For each n-unit cycle, there exists an n-periodic schedule that minimizes the long run cycle time.

Proof. This follows from Lemma 2.2 and (2.14). ∎

We observe that, for each robot move sequence, there exists a natural 'active' schedule in which the robot performs each activity as early as possible, in the order dictated by this sequence. It is not very difficult to see that the long run cycle time of an active schedule is always optimal for the corresponding robot move sequence (see also Cohen et al. [1985]). We also notice that Theorem 2.3 holds, in particular, for n-unit cycles that are the concatenation of n 1-unit cycles.

Knowing that n-periodic schedules are optimal let us now present an efficient algorithm to find an optimal n-periodic schedule. In fact, we already know how to find an optimal schedule once we have the minimum cycle time and a vertex that achieves the maximum in the right hand side of (2.15). It remains to give an algorithm to compute the minimum cycle time.

We begin by recalling that any longest cycle that goes around G at least once necessarily contains at least one vertex from the set $F = \{v_0^1, v_1^1, \ldots, v_m^1\}$. For all $0 \leq j < i \leq m$, compute the length $l_{i,j}$ of a longest path between v_i^1 and v_j^1 for each v_i^1, v_j^1 in F, that goes around G zero times. For all $0 \leq i \leq j \leq m$, compute the length $l_{i,j}$ of a longest path between v_i^1 and v_j^1 for each v_i^1, v_j^1 in F, that goes around G once. This can be done by a simple one pass label updating algorithm and it takes time $O(nm^2)$. Based on these paths we build a new digraph $G'(V', A')$ as follows. The node set $V' = F$, and the graph is complete. The arc between nodes v_i^1 and v_j^1 has length l_{ij}. Now, every simple cycle in G induces a simple cycle in G'. The inverse, however, is not necessarily true. On the other hand, a cycle in G whose length realizes the maximum in (2.15), does by definition not contain as a subcycle, a cycle that has less than maximum weight in G' (where again, the weight of a cycle is its length divided by the number of arcs of which it consists). Thus we find that a maximum weight cycle in G' must correspond to a cycle in G that realizes the maximum in (2.15). A maximum weight cycle in G' can be found in $O(m^3)$ time by the algorithm of Karp [1978]. Thus, we can find an n-periodic schedule S with minimum cycle time by the algorithm of Figure 2.4.

This algorithm has lower time complexity than the algorithms proposed by Hall et al. [1993] and Ioachim and Soumis [1995] for multiple part types.

1. For all $i, j \in \{0, \ldots, m\}$, compute the longest path from v_i^1 to v_j^1, that goes around G once if $j \geq i$ or zero times if $j < i$. Complexity: $O(nm^2)$.
2. Build G'.
3. Find a vertex v_*^1 on the maximum weight cycle, and the weight L^* of the maximum weight cycle in G'. Complexity: $O(m^3)$ time.
4. Compute $d(v_*^1, v_i^j, t)$ in G for all $0 \leq i \leq m, 1 \leq j \leq n, 0 \leq t \leq m$, by a one pass label updating algorithm that traverses G $m+1$ times. Complexity: $O(nm^2)$.
5. Compute $S(A_i^j, 0) = \max_{0 \leq t \leq m}\{d(v_*^1, v_i^j, t) - t \times L^*\}$ for all $1 \leq i \leq m+1, 1 \leq j \leq n$. Complexity: $O(nm^2)$.

Figure 2.4: Algorithm to compute a schedule with minimum cycle time

2.5 Other topics and further research

This chapter discusses cycle time minimization problems as they occur in robotic cells of different types. We have attempted to describe the state of the art of the various models and to give a classification of the problems. As yet, the most basic and important problems are more or less understood, at least from a complexity viewpoint. However, several interesting problems still remain open. In this section we overview what we think to be the most tempting areas for further research.

The most severe restrictions we placed on (certain) models consisted in considering only 1-unit and periodic schedules. As we have seen, these restrictions often allowed for efficient solution strategies, but may have excluded better solutions beforehand. Moreover, even though the description of sequences should preferably be short, there may be short sequences that are not 1-unit sequences and that are well worth considering in practical applications (see Song et al. [1993]).

Let us first overview some results that justify restrictions of the aforementioned type. The strongest result is probably the algorithm by Hall et al.[1995a] to compute simultaneously the optimal n-unit feasible robot move sequence and the optimal part input sequence in a two machine cell, where the upperbounds on the processing windows are plus infinity. The results in the previous section show that n-periodic schedules are optimal over all schedules for n-unit cycles in the same environment. Sethi et al. [1992] conjecture that in case of identical parts, 1-unit robot move sequences are optimal in a three machine cell, in case the robot travel times satisfy the triangle equality. Hertz [1995] shows that, if this equality does not hold, then the conjecture fails. Hall et

al. [1995a] showed that the conjecture holds in several special cases. Crama & Van de Klundert [1996] (see Chapter 4) prove a very general version of the conjecture. Recalling that 1-periodic schedules minimize average long run cycle time in this case, we conclude that this problem can be solved to optimality even when the 1-unit cycle restriction is relaxed. Crama & Van de Klundert [1996] in turn conjecture that 1-unit robot move sequences are optimal for an arbitrary number of machines. This would imply that the algorithm proposed by Crama & Van de Klundert [1995] (see also Chapter 3) solves such problems to optimality even if the 1-unit cyle restriction is relaxed. On the negative side, Hall et al.[1995a] show that in case of multiple part types, the set of 1-unit robot move sequences does not necessarily contain an optimal solution.

In the case of finite upperbounds on the processing windows, we also restricted our discussion to 1-periodic schedules. It is known, however, that both in the general case (Lei [1995]), and in the no-wait case (Kats [1995]), neither the set of 1-periodic schedules, nor the set of 1-unit cycles, necessarily contain an optimal solution, even in the case of identical parts.

A number of questions arise from this overview :

1. When 1-periodic schedules are not optimal, derive a tight upperbound b such that there always exists a b'-periodic schedule that achieves the minimum long run cycle time, with $b' \leq b$.

2. When 1-periodic schedules are not optimal, derive a (tight) bound on their relative performance (e.g. worst case ratio).

3. Devise an algorithm to find a periodic schedule of low periodicity that minimizes the long run cycle time.

4. When 1-unit cycles are not optimal, derive a tight upperbound c, such that there always exists an optimal k-unit cycle, with $k \leq c$.

5. When 1-unit cycles are not optimal, derive a tight bound on their relative performance.

6. Devise an algorithm to find a k-unit cycle that minimizes the long run cycle time.

Crama & Van de Klundert [1996b] present a disjunctive graph model for robot move sequence optimization. At this point, however, it is not clear how this model could be of aid in sequence optimization problems. The authors show how this model too can be used to prove the aforementioned conjecture. Furthermore, based on this model, it is possible to derive a recurrence relation for the number of k-unit cycles in a m machine robotic cell. The reader may check that, in contrast to Theorem 2.1, for $k > 1$, a permutation of the activities in which each activity occurs k times, is not necessarily a feasible robot move sequence (e.g. $A_0, A_0, A_1 \ldots$). A necessary and sufficient feasibility condition is that, between every two executions of A_i, A_{i+1} be executed exactly once,

for $i = 0, \ldots, m-1$. Since in a feasible robot move sequence every machine is unloaded as often as it is loaded, this also implies that between every two occurences of A_{i+1}, A_i occurs once. In terms of the machines, this simply states that between two consecutive loadings the machine should be unloaded, and, vice versa, between two consecutive unloadings the machine should be loaded.

The results in Section 2.3 leave open the complexity of robotic cell scheduling problems with a constant number of part types. Notice, that this does not imply that the number of parts is also bounded from above by a constant. However, in this case, the description of the input only requires an amount of space that is logarithmic in the number of parts. Hence, any polynomial algorithm for such problems should have running time that is only logarithmic in the number of parts. At this moment, there are no specialized algorithms available for such problems, even if we allow them to be polynomial in the number of parts. In practice, it may often be the case that the number of parts in the minimal part set is small. Therefore, algorithms whose complexity is exponential in the number of part types, and polynomial in (the logarithm of) the number of parts may still be of practical interest. Problems of this type also pose additional difficulties in ordinary flowshop scheduling problems. Cosmadakis & Papadimitrou [1984], and Van de Klundert [1995] provide algorithms for no-wait flowshop problems with three or more machines. Agnetis [1989] studies the case of two machines, in which the no-wait and the no-buffer problem are identical. He provides an adaptation of the Gilmore and Gomory algorithm that is polynomial in the number of part types. The no-buffer problem with three or more machines is completely open.

Chapter 3

Scheduling of identical parts in a robotic flowshop

3.1 Introduction

Recently, scheduling problems arising in flexible manufacturing cells, flexible flowlines and similar automated production systems have received much attention in the literature. In such environments, transportation of the parts between the machines is usually performed by an automated material handling system, be it a conveyor, or a pool of automatically guided vehicles (AGVs), or a robot. Much of the scheduling literature, however, has ignored the constraints placed by material handling devices on the efficiency of the productive system, either because these devices were not regarded as bottlenecks, or, more pragmatically, for reasons of modeling simplicity. Only recently has material handling been paid special attention and been incorporated explicitly in scheduling models (see e.g. Blazewicz et al. [1991], Hall et al. [1995a, 1995b], Hall et al. [1995c], Jeng et al. [1993], King et al. [1992], Kise [1991], Kise et al. [1991], Sethi et al. [1992]). An overview of such scheduling problems is given in Chapter 2.

In this chapter, we investigate a cyclic scheduling problem for a robotic flowshop whose throughput rate is highly dependent on the interaction between the material handling system (namely, the robot) and the machines. More precisely, we consider a robotic flowshop consisting of m machines, an input device, an output device and a robot (see Figures 3.1 and 3.2). There are no buffers in the flowshop (a similar problem with buffers is considered in King et al. [1993]). Transportation of the parts between the machines is taken care of by the robot, which can only handle one part at a time. In the most general setting of the problem, a so-called minimal part set (MPS) is to be repeatedly produced, where the MPS consists of parts of different types in proportion to a certain target production mix (see e.g. Stecke [1983]). The objective of the scheduling problem is then to determine the part input sequence (i.e., the order in which the parts in the MPS should be processed) and the corresponding sequence of robot moves so as to maximize the long run throughput rate or to minimize the long run cycle time of the system.

This problem (and closely related ones) has been considered by several authors (Sethi et al. [1992] and Hall et al. [1995a] provide references). Sethi et al. [1992] showed that, when there are only two machines (and under some restrictions on the move sequences that the robot is allowed to perform), the problem can be solved in polynomial time. The same result was obtained by Kise et al. [1991] for a makespan minimization objective. On the other hand, Hall et al. [1995b] proved that the problem is already strongly NP-hard for a three-machine robotic flowshop. As a matter of fact, these authors established that computing the optimal part input sequence in a three-machine flowshop is strongly NP-hard, even when the robot move sequence is given. A further classification of the complexity of special cases in which the robot move sequence is fixed can be found in Sriskandarajah et al. [1995].

In our work, by contrast, we restrict ourselves to the special case of the problem where the number of machines is arbitrary, but all parts are of the same type. In this framework, the part input sequencing problem vanishes altogether and the term *cyclical*, that usually indicates in the literature that the part input sequence repeats identically for each and every MPS (see e.g. Agnetis et al. [1993], Karabati & Kouvelis [1993], McCormick et al. [1989]), applies here only to the sequence of moves performed by the robot.

The resulting *identical parts cyclic scheduling problem* has been investigated by Sethi et al. [1992] and Hall et al. [1995a]. More precisely, in the classification scheme of Hall et al. [1995a], we are interested in the problem R$Cm|k = 1$,1-unit$|C_t$, meaning that the robotic cell contains m machines, that there is exactly one part type, and that the objective is to minimize the cycle time C_t under the restriction that one unit be produced in each cycle. In particular, Sethi et al. [1992] described a simple decision rule that computes the optimal robot move sequence when there are only three machines in the flowshop. In this chapter, we considerably extend their analysis by proving that the identical parts cyclic scheduling problem can be solved in time polynomial in m, where m denotes the number of machines in the shop.

In Section 3.2, we give a more precise definition of the identical parts cyclic scheduling problem, and we describe a one-to-one correspondence (discovered by Lieberman & Turksen [1981], Sethi et al. [1992]) between its feasible solutions and the permutations of the set $\{1, \ldots, m\}$ (see also Theorem 2.1). In Section 3.3, we derive upper and lower bounds on the optimal cycle time. We also present in this section the key result of this chapter, namely that the set of pyramidal permutations necessarily contains an optimal solution of the problem (pyramidal permutations have been previously introduced in the framework of the traveling salesman problem; see e.g. Gilmore et al. [1985]). In Section 3.4, we give an efficient algorithm to compute the cycle time of a schedule described by a pyramidal permutation. Relying on this result, we present in Section 3.5 a dynamic programming approach that allows to solve the recognition version of the identical parts cyclic scheduling problem in $O(m^2)$ time, and its optimization version in $O(m^3)$ time. Finally, we discuss in Section 3.6 some directions for further research.

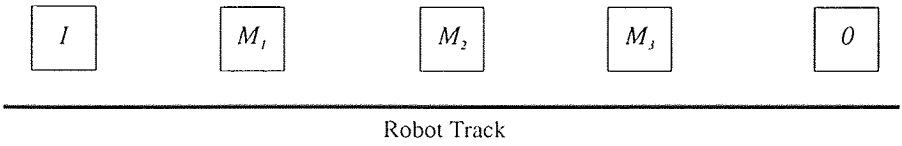

Figure 3.1: A 3-machine robotic cell (line layout)

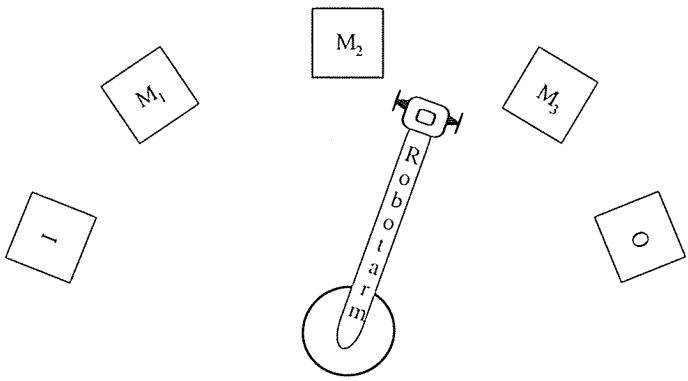

Figure 3.2: A 3-machine robotic cell (circle layout)

3.2 Cycles, permutations & schedules

In this section we discuss the input parameters of the problem and its objective. A solution for the problem is defined as a sequence of robot moves that maximizes the long run throughput rate. The problem is shown to be a permutation problem. Furthermore, the objective of the problem is restated in terms of schedules and cycle times, rather than throughput rates.

Let us first define the notation we use for the entities that play a role in the problem. The m machines of the robotic cell are denoted by $M_1 \ldots M_m$. The input device is denoted by I or M_0. The output device is denoted by O or M_{m+1}. Each part is initially available at the input device and must be processed succesively by $M_1, M_2, \ldots, M_m$, until it is unloaded at the output device. Each machine can only process one part at a time and there are no buffers for intermediary storage at the machines. We denote the processing time of the part on machine M_i by p_i, $i = 1 \ldots m$. We call the segment of the robot track between two adjacent machines a trajectory, and we denote by δ_i the time the robot needs to travel from machine M_i to M_{i+1} , or from M_{i+1} to M_i, $i = 0, \ldots, m$. Loading a part onto $M_i, i = 1, \ldots, m+1$ or unloading a part from $M_i, i = 0, \ldots, m$ takes time ϵ_i. Hence the input of the problem consists of:

- processing times $p_1, \ldots, p_m$
- travel times $\delta_0, \ldots, \delta_m$
- (un)loading times $\epsilon_0, \ldots, \epsilon_{m+1}$

For reasons of clarity we usually assume $\delta_i = \delta, i = 0, \ldots, m$, $\epsilon_i = \epsilon, i = 0, \ldots, m+1$. However, all results presented go through for trajectory and machine dependent travel and (un)loading times.

Let us now describe the type of robot moves that we want to consider. From a practical viewpoint it is not desirable to specify all moves the robot has to perform until a complete batch is processed, since the batch size may be fairly large (we assume it to be infinite). Hence we will be interested in more compact sequences that the robot can execute a number of times. More precisely, we will be interested in sequences with the property that exactly one part is taken from the input device (and one part is dropped at the output device) in each execution of the sequence. Such sequences of robot moves are called 1-unit cycles :

Definition 3.1 A *1-unit cycle* is a sequence of robot moves in which each machine is loaded and unloaded exactly once.

Observe that a 1-unit cycle returns the cell in its original state and hence can be repeated infinitely many times. Sethi et al. [1992] conjecture that the maximum throughput rate which can be achieved executing a 1-unit cycle equals the maximum

throughput rate over all sequences of robot moves. A weak form of this conjecture has been proved by Shriskandarajah et al. [1995] for the identical parts 3-machine cyclic scheduling problem. Chapter 4 establishes that a slightly weaker form of the conjecture holds. The conjecture provides further motivation for restricting our attention to 1-unit cycles.

Lieberman & Turksen [1981], and Sethi et al. [1992] have the following theorem on the number of possible 1-unit cycles in a robotic cell with m-machines (see also Theorem 2.1):

Theorem 3.1 (Sethi et al.) In a robotic cell with m machines, there are exactly $m!$ 1-unit cycles.

The following definition is helpful to understand Theorem 3.1.

Definition 3.2 For all $i, i = 0 \ldots, m$, *activity* A_i consists of the following sequence of robot moves :

1. unload M_i
2. travel from M_i to M_{i+1}
3. load M_{i+1}

Without loss of generality, it may be assumed that every 1-unit cycle starts with the robot moves as specified by A_0. The proof of Theorem 3.1 establishes that every 1-unit cycle defines a permutation of the activities starting with A_0 and, conversely, that every permutation of the activities starting with A_0 corresponds to a 1-unit cycle. Thus, computing an optimal 1-unit cycle is equivalent to computing an optimal permutation of the activities. In the sequel, we will use the names "1-unit cycle" and "permutation of the activities" interchangeably.

Let us now concentrate on the objective function of our problem. Informally speaking, we want to maximize the long run average throughput rate of the system, or equivalently, we want to minimize its long run average cycle time. To make this concept more precise, consider the following definitions.

Definition 3.3 A *schedule* is a function $S(A_i, t)$ that assigns a starting time to the t-th execution of activity A_i ($i = 0, \ldots, m,\ t \in \mathbb{N}$). The *long run average cycle time* of S is equal to

$$\lim_{t \to \infty} \frac{S(A_m, t)}{t}$$

Definition 3.4 A schedule S is called a *1-periodic schedule* if there exists a constant L (called the *cycle time* of S) such that for every $A_i, i = 0, \ldots m$, and for every $t \in \mathbb{N}, S(A_i, t+1) - S(A_i, t) = L$.

Definition 3.5 Given a permutation of the activities, say $\pi = (A_{i_0}, A_{i_1}, \ldots, A_{i_m})$, and a schedule $S(A_i, t)$, we say that S is *a schedule for* π if the sequence of activities defined by S is consistent with π, i.e. $S(A_{i_j}, t) < S(A_{i_k}, t)$ for all $j, k \in \{0, \ldots, m\}$ with $j < k$ and for all $t \in \mathbb{N}$.

Clearly, for a 1-periodic schedule, the long run average cycle time coincides with the cycle time. In Chapter 2 we have seen that, for each 1-unit cycle, there exists a 1-periodic schedule S that minimizes the long run average cycle time over all schedules. (This conclusion could also be drawn from an analysis of the periodical behavior of the cell, viewed as a discrete system; see e.g. Cohen et al [1985], Sethi et al. [1992].)

Definition 3.6 Let π be a permutation of the activities. The *cycle time* of π, denoted $L(\pi)$, is the minimum cycle time achievable by a 1-periodic schedule for π.

We observe here that the computation of the cycle time of a fixed permutation of the activities can be formulated as the solution of a linear programming model, similar to the one used in critical path methods (see Chapter 2). Some of the proofs to come (e.g. Lemma 3.2 and Theorem 3.4) could be recast entirely in this LP framework.

With these definitions at hand, we can formulate as follows the (optimization version of the identical parts cyclic scheduling problem :

Definition 3.7 *Identical Parts Cyclic Scheduling Problem*

INPUT : processing times $p_1, \ldots, p_m$, travel times $\delta_0, \ldots, \delta_m$ and (un)loading times $\epsilon_0, \ldots, \epsilon_{m+1}$,

QUESTION : find a permutation of the activities with minimum cycle time.

3.3 Pyramidal permutations.

In this section, we first give a lower bound on the cycle time of the optimal permutation, and we describe a permutation whose cycle time never exceeds twice the lower bound. These results and their derivation may help the reader gain some intuition for the problem, and will also play a role in the analysis presented in Sections 3.4 and 3.5. In the second part of the section, we introduce pyramidal permutations and show that the set of pyramidal permutations necessarily contains an optimal 1-unit cycle.

Lemma 3.1 The cycle time $L(\pi)$ of every permutation π satisfies:

$$L(\pi) \geq \max\{2(m+1)(\delta+\epsilon), \max_i p_i + 4(\delta+\epsilon)\}$$

Proof. Consider a permutation π of the activities and assume without loss of generality that π starts with A_0. Since the next cycle starts again with A_0, in any cycle the robot must at least travel from I to O and back to I, which induces a travel time of at

least $2\delta(m+1)$. Also, in any cycle every machine must be loaded and unloaded, the input must be unloaded and the output must be loaded; hence the total time the robot spends loading and unloading machines is at least $2\epsilon(m+1)$. Thus we have that $L(\pi) \geq 2(m+1)(\delta+\epsilon)$.

To prove that $L(\pi) \geq \max_i p_i + 4(\delta+\epsilon)$, fix $i \in \{1,\ldots,m\}$, and consider an optimal 1-periodic schedule for π, say S. Then, $L(\pi) = S(A_i, t+1) - S(A_i, t)$, i.e. the cycle time equals the time between two consecutive unloading operations of machine M_i. Now consider the point in time τ between $S(A_i,t)$ and $S(A_i,t+1)$ at which M_i starts processing. Between $S(A_i,t)$ and τ, the robot must at least have performed A_i and A_{i-1}. Hence we have $\tau \geq S(A_i,t) + 4\delta + 4\epsilon$. Furthermore, the unloading operation starting at $S(A_i,t+1)$ cannot be performed before machine M_i has finished processing the part, i.e. $S(A_i,t+1) \geq \tau + p_i$. From these two inequalities we deduce $L(\pi) \geq p_i + 4\delta + 4\epsilon$, which concludes the proof. ∎

If the robot is relatively slow, its travel time is likely to be the bottleneck of the system. In this case, the permutation $A_0, A_1, \ldots, A_m$, to be called π_U, might well be the optimal permutation since it has minimum travel time. On the other hand, if the robot is relatively fast, the permutation $A_0, A_m, A_{m-1}, \ldots, A_1$, to be called π_D, appears to be a good alternative, since it allows each machine as much time for processing as possible. We now derive an expression for $L(\pi_D)$:

Lemma 3.2

$$L(\pi_D) = \max\{4m\delta + 2(m+1)\epsilon, \max_i p_i + 4(\delta+\epsilon)\}$$

Proof. The total travel time, and load/unload time for π_D is equal to $4m\delta + 2(m+1)\epsilon$ and is a lowerbound for $L(\pi_D)$. By Lemma 3.1 we know that $L(\pi_D) \geq \max_i p_i + 4\delta + 4\epsilon$. Thus the maximum over these two is a lowerbound for $L(\pi_D)$. Let C equal this maximum. We give a schedule for π_D with cycle time C and prove its feasibility by induction. Observe that a schedule is feasible if the robot can indeed reach every machine in time, and never unloads a machine before it has finished processing. For notational convenience, we shift π_D and write $\pi_D = (A_m, A_{m-1}, \ldots, A_0)$.

Let $S(A_i,t) = (t-1)C + (m-i)(2\epsilon + 3\delta)$, for $i = 0,\ldots,m$ and $t \in \mathbb{N}$.

We are now going to complete the proof of the Lemma by showing that $S(A_i,t)$ is a feasible schedule. We proceed by forward induction on t and backward induction on $i = m, m-1, \ldots, 0$. Assume that at the start of the first cycle all machines are loaded and have finished processing their part (this is without loss of generality, since we are only interested in the long run behavior of the system). For $t = 1$ and $i = m$, $S(A_m,1) = 0$. For $t = 1$ and $i < m$, $S(A_i,1) = (m-i)(2\epsilon + 3\delta)$, which is precisely the time required for the robot to perform $A_m, \ldots, A_{i+1}$, and to reach M_i.

Fix $t > 1$ and $i = m$; by induction, the robot arrives at M_m at time

$$\begin{aligned} S(A_0, t-1) + \epsilon + \delta + \epsilon + (m-1)\delta &= (t-2)C + m(2\epsilon + 3\delta) + 2\epsilon + m\delta \\ &\leq (t-2)C + C \\ &= (t-1)C \\ &= S(A_m, t). \end{aligned}$$

Thus the robot can reach M_m in time to perform A_m in the t-th cycle. In the previous cycle, the robot finished loading machine M_m at time

$$l(m, t-1) = S(A_{m-1}, t-1) + \epsilon + \delta + \epsilon.$$

We have:

$$\begin{aligned} S(A_m, t) - l(m, t-1) &= C - 2\epsilon - 3\delta - \epsilon - \delta - \epsilon \\ &= C - 4\epsilon - 4\delta \\ &\geq p_m. \end{aligned}$$

Thus machine M_m has finished processing the part at time $S(A_m, t)$ and can be unloaded.

Now, for $t > 1, i < m$: by induction, the robot starts unloading machine M_{i+1} at time $S(A_{i+1}, t)$. It then arrives at machine M_i at time $S(A_{i+1}, t) + \epsilon + \delta + \epsilon + 2\delta = S(A_i, t)$. In the previous cycle, it finished loading machine M_i at time $l(i, t-1) = S(A_{i-1}, t-1) + \epsilon + \delta + \epsilon$. This yields that

$$\begin{aligned} S(A_i, t) - l(i, t-1) &= C - 2\epsilon - 3\delta - \epsilon - \delta - \epsilon \\ &= C - 4\epsilon - 4\delta \\ &\geq p_i. \end{aligned}$$

Thus machine M_i has indeed finished processing at time $S(A_i, t)$, and the robot may start unloading. ■

Theorem 3.2 The optimal permutation π is such that:

$$\max\{2(m+1)(\delta+\epsilon), \max_i p_i + 4(\delta+\epsilon)\} \leq L(\pi)$$

$$L(\pi) \leq \max\{4m\delta + 2(m+1)\epsilon, \max_i p_i + 4(\delta+\epsilon)\}.$$

Proof. The bounds follow from Lemma 3.1 and Lemma 3.2. ■

Incidentally, Theorem 3.2 implies that the cycle time of π_D is always smaller than twice the optimal cycle time. In other words, the algorithm that outputs π_D, independently of the values of the input parameters, is a 2-approximation algorithm for

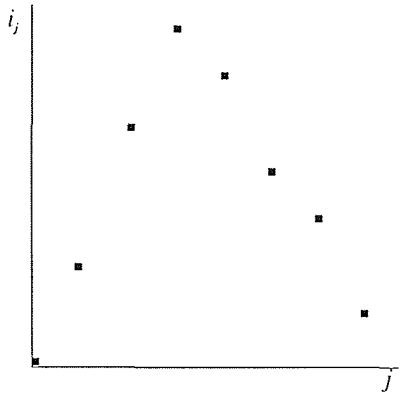

Figure 3.3: The pyramidal permutation $A_0, A_2, A_5, A_7, A_6, A_4, A_3, A_1$

the identical parts cyclic scheduling problem! (We will not make use of this observation, but we find it interesting in its own right.) Moreover, π_D is optimal when $L(\pi_D) = \max_i p_i + 4(\delta + \epsilon)$. This provides an important proviso for the (unmotivated) claim made by Asfahl [1985, p. 274] that the permutation π_D 'must be held regardless of the relationship between the machine cycle times, the time required for the robot to move from station to station, and the load/unload times'. (the author calls 'machine cycle time' what we call 'processing time'.)

Definition 3.8 A set of permutations Π is *dominating* if for every choice of the processing times, there exists $\pi \in \Pi$ such that $L(\pi) \leq L(\pi')$ for all $\pi' \notin \Pi$.

We are now going to introduce a class of permutations, of which π_U and π_D are just two special representatives, and we are going to show that this class is dominating.

Let $\pi = (A_0, A_{i_1}, \ldots, A_{i_k}, A_{i_{k+1}}, \ldots, A_{i_m})$.

Definition 3.9 π is *pyramidal* if $1 \leq i_1 < \ldots < i_k = m$ and $m > i_{k+1} > \ldots > i_m \geq 1$.

In particular, the permutations π_U and π_D are pyramidal. The meaning of the adjective pyramidal should become clear from Figure 3.3. It is probably worth noticing that the concept of pyramidal permutations is not new : it has been introduced earlier, and extensively studied, in the literature on the Traveling Salesman problem; see Gilmore et al. [1985] for a thorough account, as well as Section 3.5 below. For an arbitrary, not necessarily pyramidal, permutation we also define:

Definition 3.10 Activity A_{i_k} is *uphill pyramidal* if there is an index l in $\{k, \ldots, m\}$ such that $i_k < i_j$ for all $k < j \leq l$, and $i_k > i_j$ for all $j < k$ and all $j > l$.

In words : all activities between A_{i_k} and A_{i_l} bear on machines located after M_{i_k} in the flowshop, while all activities before A_{i_k} or after A_{i_l} bear on machines located before M_{i_k}.

Definition 3.11 Activity A_{i_k} is *downhill pyramidal* if there is an index l in $\{0, \ldots, k\}$ such that $i_j > i_k$ for all $l \leq j < k$ and $i_j < i_k$ for all $j < l$ and all $j > k$.

In words : all activities between A_{i_l} and A_{i_k} occur on machines located after M_{i_k} in the flowshop, while all activities before A_{i_l} or after A_{i_k} occur on machines located before M_{i_k}.

Remarks.

1. A_0 and A_m are uphill pyramidal and A_m is downhill pyramidal in all permutations.

2. A permutation is pyramidal if and only if each activity is pyramidal (i.e. either uphill or downhill pyramidal) in this permutation.

3. The reader should convince himself that A_{i_k} is uphill pyramidal if and only if the trajectory $[M_{i_k}, M_{i_k+1}]$ is travelled exactly twice by the robot in each cycle : once when performing A_{i_k} and once after performing A_{i_l}.

4. Similarly, except for $i_k = m$, A_{i_k} is downhill pyramidal if and only if the trajectory $[M_{i_k}, M_{i_k+1}]$ is travelled exactly four times in each cycle : once just before A_{i_l}, once just before A_{i_k}, once during A_{i_k}, and once just after A_{i_k}.

The following theorem justifies our interest in pyramidal permutations. It will be the cornerstone for all subsequent results, and can therefore be viewed as the main result in this chapter.

Theorem 3.3 The set of pyramidal permutations is dominating.

Proof. For reasons of clarity, and to stress that the theorem holds under very general conditions, we present the proof for the case where the machines are not necessarily equidistant, and loading/unloading times are machine dependent. We first introduce the following notations : for all $i, j = 0, \ldots, m$,

$$\delta_{ij} = \begin{cases} \sum_{k=i}^{j} \delta_k & \text{if } i \leq j \\ \\ \sum_{k=j}^{i} \delta_k & \text{if } j \leq i. \end{cases}$$

The time the robot takes to perform A_i is denoted by Δ_i :

$$\Delta_i = \epsilon_i + \delta_i + \epsilon_{i+1}.$$

Similarly to δ_{ij}, we define Δ_{ij} as:

$$\Delta_{ij} = \begin{cases} \sum_{k=i}^{j} \Delta_k & \text{if } i \leq j \\ \sum_{k=j}^{i} \Delta_k & \text{if } j \leq i. \end{cases}$$

Let π be a nonpyramidal permutation. Let A_q be a nonpyramidal activity, let $A_{i_r} = A_b$ be the uphill pyramidal activity defined by $b = \max\{j | j < q \text{ and } A_j \text{ is uphill pyramidal }\}$, and let $A_{i_s} = A_e$ be the uphill pyramidal activity defined by $e = \min\{j | j > q \text{ and } A_j \text{ is uphill pyramidal }\}$.

Since A_{i_s} and A_{i_r} are uphill pyramidal, there exist indices i_l (associated with A_{i_s} as in Definition 3.10) and i_{k-1} (associated with A_{i_r} as in Definition 3.10) such that π can be rewritten in the form :

$$\pi = (A_0, \ldots, A_{i_r}, A_{i_{r+1}}, \ldots, A_{i_{s-1}}, A_{i_s}, \ldots, A_{i_l}, A_{i_{l+1}}, \ldots, A_{i_{k-1}}, A_{i_k}, \ldots, A_{i_m})$$

and

- all activities in $\pi_1 = (A_0, \ldots, A_{i_r})$ bear on machines with index at most $b+1$, i.e. $i_j \leq i_r = b$ for all A_{i_j} in π_1 (since A_{i_r} is uphill pyramidal),
- for all A_{i_j} in $\pi_2 = (A_{i_{r+1}}, \ldots, A_{i_{s-1}}), i_r = b < i_j < i_s = e$ (since A_{i_s} is uphill pyramidal),
- for all A_{i_j} in $\pi_3 = (A_{i_s}, \ldots, A_{i_l}), i_j \geq i_s = e$ (by definition of i_l),
- for all A_{i_j} in $\pi_4 = (A_{i_{l+1}}, \ldots, A_{i_{k-1}}), i_r = b < i_j < i_s = e$ (by definition of i_l and i_k),
- for all A_{i_j} in $\pi_5 = (A_{i_k}, \ldots, A_{i_m}), i_j < b$ (by definition of i_k).

Notice that π_1 and π_3 can never be empty since A_0 and A_m are uphill pyramidal by definition. Since there exists a nonpyramidal activity A_q, $\pi_2 \cup \pi_4$ cannot be empty, although one of π_2 or π_4 can. Finally notice that π_5 can be empty.

We claim that π is dominated by the new permutation

$$\pi' = \pi_1, \pi_3, A_{e-1}, A_{e-2}, \ldots, A_{b+1}, \pi_5,$$

i.e.

$$L(\pi') \leq L(\pi).$$

Before proving this claim, notice that the status (pyramidal or nonpyramidal) of all activities contained in π_1, π_3, π_5 is the same in π' as in π, and that all activities contained in $\pi_2 \cup \pi_4$, i.e. $A_{e-1}, \ldots, A_{b+1}$ are downhill pyramidal in π' (Figure 3.4 gives a sketchy representation of the permutation π', where thick lines indicate the segments π_1, π_3, π_5 that π' inherits from π). Thus the claim implies that, in at most m iterations, π can

be transformed into a pyramidal permutation whose cycle time is no larger than that of π, which establishes Theorem 3.3.

Let a 1-periodic schedule with minimum cycle time for π be given by $S(A_i, t)$. Denote by $l(i,t)$ the time at which the robot ends loading M_i in the *t-th* execution of the 1-unit cycle, for all $t \geq 1$, when it performs schedule S. We give now a 1-periodic schedule $T(A_i, t)$ for π' such that $T(A_0, t) = S(A_0, t)$, thereby showing that the cycle time of π' is at most $L(\pi)$. We denote by $\lambda(i,t)$ the time at which the robot ends loading M_i in the *t-th* execution of the 1-unit cycle, for all $t \geq 1$, when it performs schedule T.

For all $t \geq 1$, we let

$$T(A_j, t) = S(A_j, t) \qquad \text{if } 0 \leq j \leq b \tag{3.1}$$

Next, for all $t \geq 1$ we define $T(A_{b+1}, t)$ by

$$\begin{aligned} T(A_{b+1}, t) &= T(A_{i_k}, t) - \Delta_{b+1} - \delta_{(b+1)i_k} && \text{if } \pi_5 \neq \emptyset \quad (3.2)\\ &= T(A_0, t+1) - \Delta_{b+1} - \delta_{(b+1)0} && \text{otherwise,} \quad (3.3) \end{aligned}$$

and, recursively on j :

$$T(A_j, t) = T(A_{j-1}, t) - \Delta_j - \delta_{j(j-1)} \qquad \text{if } b+1 < j < e. \tag{3.4}$$

Finally , we let

$$T(A_{i_l}, t) = T(A_{e-1}, t) - \Delta_{i_l} - \delta_{i_l(e-1)} \tag{3.5}$$

and

$$T(A_j, t) = S(A_j, t) + T(A_{i_l}, t) - S(A_{i_l}, t) \qquad \text{if } e \leq j \leq m. \tag{3.6}$$

Notice that the definition is complete, i.e. $T(A_j, t)$ is defined for all $t \geq 1$ and for all $j \in \{0, \ldots, m\}$. In particular, (3.1) applies to π_1 and π_5 , (3.2)-(3.4) apply to π_2 and π_4 and (3.5)-(3.6) apply to π_3. One also checks easily that schedule T is 1-periodic, with cycle time $L = L(\pi)$.

To prove that (3.1)-(3.6) define a feasible schedule for π', we need to check that:

1. the robot can reach M_j before $T(A_j, t)$ in cycle t,
2. machine M_j has finished processing a part at time $T(A_j, t)$ in cycle t.

We first prove that the robot can reach all machines in time in every cycle. Consider any activity A_j, and let A_l be the activity preceding A_j in π'. If the start-time of A_l is defined by one of (3.2)-(3.5) (i.e. if $j \in \{b+1, \ldots, e-1\} \cup \{i_k\}$), then $T(A_j, t) - T(A_l, t)$ is exactly the time required for the robot to perform A_l (viz. Δ_l) and to subsequently move from M_{l+1} to M_j (viz. δ_{lj}). Thus the robot can get to M_j at time $T(A_j, t)$ if it can get to M_l at time $T(A_l, t)$.

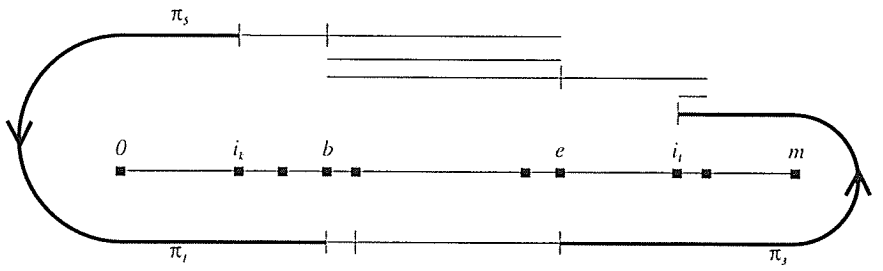

Figure 3.4: Graphical representation of the permutation π'

The latter conclusion also applies if $0 \leq j \leq b, j \neq i_k$, in view of (3.2) and if $e < j \leq m$, in view of (3.6) (since the schedule S is feasible).

This reasoning leaves only open the question whether the robot can reach M_e at time $T(A_e, t)$ given that it starts with A_b (the activity preceding A_e in π') at time $T(A_b, t)$. Thus we have to check that

$$T(A_b, t) + \Delta_b + \delta_{(b+1)(e-1)} \leq T(A_e, t).$$

From the fact that in a schedule for π every trajectory $[M_j, M_{j+1}], b < j < e$ is travelled at least four times we can derive that (see Figure 3.4.):

$$\begin{aligned} S(A_b, t+1) - S(A_b, t) \;\geq\; & S(A_b, t+1) - S(A_{i_k}, t) + S(A_{i_l}, t) - S(A_e, t) \\ & + \delta_{bi_k} + \delta_{i_l e} + \Delta_{i_l} + 3\delta_{(b+1)(e-1)} + \Delta_{(b+1)(e-1)} + \Delta_b. \end{aligned}$$

Combining this with (3.1) and (3.6) gives

$$\begin{aligned} T(A_b, t+1) - T(A_b, t) \;\geq\; & T(A_b, t+1) - T(A_{i_k}, t) + T(A_{i_l}, t) - T(A_e, t) \\ & + \delta_{bi_k} + \delta_{i_l e} + \Delta_{i_l} + 3\delta_{(b+1)(e-1)} + \Delta_{(b+1)(e-1)} + \Delta_b. \end{aligned}$$

Rewriting this inequality, we get

$$\begin{aligned} T(A_e, t) \;\geq\; & T(A_b, t) - T(A_{i_k}, t) + T(A_{i_l}, t) \\ & + \delta_{(b+1)i_k} + \delta_{i_l e} + \Delta_{i_l} + 3\delta_{(b+1)(e-1)} + \Delta_{(b+1)(e-1)} + \Delta_b, \end{aligned}$$

Combining this with (3.2) and (3.4) leads to

$$\begin{aligned} T(A_e, t) \;\geq\; & T(A_b, t) - T(A_{e-1}, t) + T(A_{i_l}, t) \\ & + \delta_{i_l(e-1)} + \Delta_{i_l} + \delta_{(b+1)(e-1)} + \Delta_b. \end{aligned}$$

and thus by (3.5)

$$T(A_e, t) \;\geq\; T(A_b, t) + \delta_{(b+1)(e-1)} + \Delta_b$$

as required.

Remark Notice that we used A_{i_k}, which may not exist if π_5 is empty. In this case the result can be obtained similarly using $S(A_0, t+1)$ instead of $S(A_{i_k}, t)$.

Now we prove that machine M_j has indeed finished processing at time $T(A_j, t)$. By (1), all machines M_j with A_j in π_1 or π_5 are ready at time $T(A_j, t)$. Consider now machine M_{b+1}. Observe that the start of activity A_{b+1} in schedule T occurs as late as possible under the constraint that $S(A_{i_k}, t) = T(A_{i_k}, t)$ (see (3.1)-(3.3) and Figure 3.4). Thus, one derives that

$$S(A_{b+1}, t) \leq T(A_{b+1}, t)$$

and

$$T(A_{b+1}, t) - T(A_b, t) \geq S(A_{b+1}, t) - S(A_b, t).$$

Since $S(A_{b+1}, t)$ is feasible, we have that

$$T(A_{b+1}, t) - T(A_b, t) \geq S(A_{b+1}, t) - S(A_b, t) \geq p_{b+1} + \Delta_b,$$

as required.

A straightforward extension of the argument used in Lemma 3.1 shows that

$$p_j + \Delta_j + \Delta_{j-1} + \delta_j + \delta_{j-1} \leq L, \text{ for all } j \in \{0, \ldots, m\}.$$

Thus, for all $b+1 < j < e$,

$$\begin{aligned} \lambda(j,t) &= T(A_{j-1}, t) + \Delta_{j-1} \\ &= T(A_j, t) + \Delta_j + \delta_j + \delta_{j-1} + \Delta_{j-1} && \text{(by (3.4))} \\ &= T(A_j, t+1) + \Delta_j + \delta_j + \delta_{j-1} + \Delta_{j-1} - L, \end{aligned}$$

and thus

$$T(A_j, t+1) - \lambda(j,t) = L - (\Delta_j + \delta_j + \delta_{j-1} + \Delta_{j-1}) \geq p_j.$$

This is the required inequality : since A_j is a downhill activity, $T(A_j, t+1) - \lambda(j,t)$ represents the time elapsed between loading of a part in cycle t and its unloading in cycle $t+1$.

In view of (3.6), the machines A_j with $j > e$ create no problem. Finally, we have to check that M_e has finished processing in time:

$$T(A_e, t) - \lambda(e, t-1) =$$

$$T(A_e, t) - (T(A_{e-1}, t-1) + \Delta_{e-1}) =$$

$$T(A_e, t) - (T(A_{i_l}, t-1) + \Delta_{i_l} + \delta_{i_l(e-1)} + \Delta_{e-1}) = \quad \text{(by (1.5))}$$

$$S(A_e, t) - (S(A_{i_l}, t-1) + \Delta_{i_l} + \delta_{i_l(e-1)} + \Delta_{e-1}). \quad \text{(by (3.6))}$$

Now, there are two cases.

1. If A_e precedes A_{e-1} in π (and thus the part loaded onto M_e in each execution of π is unloaded in the next execution):

$$S(A_e,t) - (S(A_{i_l},t-1) + \Delta_{i_l} + \delta_{i_l(e-1)} + \Delta_{e-1}) \geq$$

$$S(A_e,t) - (S(A_{e-1},t-1) + \Delta_{e-1}) =$$

$$S(A_e,t) - l(e,t-1),$$

and hence the feasibility of $T(A_e,t)$ follows from the feasibility of $S(A_e,t)$.

2. If A_{e-1} precedes A_e in π (and thus the part loaded onto M_e in each execution of π is unloaded in the same execution), it is not hard to see, by just checking the travel time that

$$S(A_{e-1},t) \geq S(A_{i_l},t-1) + \Delta_{i_l} + \delta_{i_l(e-1)}.$$

Hence

$$\begin{aligned} T(A_e,t) - \lambda(e,t-1) &\geq S(A_e,t) - (S(A_{e-1},t) + \Delta_{e-1}) \\ &= S(A_e,t) - l(e,t), \end{aligned}$$

and again the feasibility of $T(A_e,t)$ follows from the feasibility of $S(A_e,t)$. ∎

We remark that, when $m = 3$, there are exactly 4 pyramidal permutations, which have been proved by Sethi et al.[1992] to be dominating. Theorem 3.3 generalizes this result for arbitrary values of m.

3.4 An algorithm for computing the cycle time of a pyramidal permutation.

In this section we present an algorithm that computes a shortest 1-periodic schedule for a pyramidal permutation in $O(m)$ time. This time complexity improves on the time complexity of the algorithm using the max-algebra approach Cohen et al. [1985], Karp [1978], and on an algorithm presented in Chapter 2, which was also based on Karp [1978] (of course, the scope of our algorithm is also narrower).

While proving the correctness of the algorithm, we derive some structural properties of a shortest 1-periodic schedule for a pyramidal permutation that will turn out to be useful in the next section.

Let $\pi = (A_0, A_{i_1}, \ldots, A_{i_m})$ be a pyramidal permutation of the activities, and let U resp. D denote the index set of the uphill, resp. downhill activities in π (with

$\{m\} = U \cap D$). A formal statement of our algorithm is given in Figure 3.5. We now discuss it more informally.

The algorithm computes a start time $S(A_i)$ for each activity A_i as well as a cycle time L_S. The schedule S is then implicitely defined by the relation

$$S(A_i, t) = S(A_i) + t \times L_S \text{ for } i = 0, \ldots, m \text{ and } t \in \mathbb{N}. \tag{3.7}$$

The algorithm proceeds backwards by decreasing activity index, starting with A_m. It schedules all downhill activities without waiting time, giving the robot just enough time to travel from machine to machine between two activities. That is, if $i \in D$ and A_j is the downhill activity preceding A_i in π, then

$$S(A_i) = S(A_j) + (j + 1 - i)\delta + \delta + 2\epsilon. \tag{3.8}$$

Next, suppose that we are about to schedule an uphill activity A_i such that A_{i+1} is also uphill. Then, for every feasible schedule T, and for all $t \in \mathbb{N}$,

$$T(A_i, t) \leq T(A_{i+1}, t) - \delta - 2\epsilon - p_{i+1}, \tag{3.9}$$

and the algorithm simply sets

$$S(A_i) = S(A_{i+1}) - \delta - 2\epsilon - p_{i+1}. \tag{3.10}$$

Next consider an uphill activity A_i such that A_{i+1} is downhill. Again, in every feasible schedule T, and for all $t \in \mathbb{N}$,

$$T(A_i, t) \leq T(A_{i+1}, t) - \delta - 2\epsilon - p_{i+1}. \tag{3.11}$$

On the other hand, if A_j denotes the uphill activity following A_i in π, then we have in every feasible schedule T,

$$T(A_i, t) \leq T(A_j, t) - (j - i)\delta - 2\epsilon. \tag{3.12}$$

The algorithm takes (3.11) and (3.12) into account, and sets

$$S(A_i) = \min\{S(A_j) - (j - i)\delta - 2\epsilon, S(A_{i+1}) - \delta - 2\epsilon - p_{i+1}\}. \tag{3.13}$$

Observe that, if $S(A_i)$ is determined by the second term in the latter expression, then the difference $S(A_j) - S(A_i)$ is larger than the travel time required between A_i and A_j. In other words, the robot will have to incur some idle time before the execution of A_j.

In this way a starting time is determined for each activity. The cycle time L_S of the schedule, however, is still not determined. It can be seen that L_S must satisfy

$$S(A_0) + L_S \geq S(A_{i_m}) + (i_m + 2)\delta + 2\epsilon, \tag{3.14}$$

since otherwise, the robot cannot reach M_0 in time to start the (next) execution of A_0 after executing A_{i_m}. Moreover, by Lemma 3.1, we know that

$$L_S \geq \max_i p_i + 4(\delta + \epsilon). \tag{3.15}$$

Finally, consider any uphill activity A_i such that A_{i-1} is downhill. The part loaded on M_i in the t-th execution of A_{i-1} is unloaded from M_i in the $(t+1)$-st execution of A_i. Hence,

$$S(A_i) + L_S \geq S(A_{i-1}) + \delta + 2\epsilon + p_i. \tag{3.16}$$

In the algorithm, L_S is set to the minimum value that satisfies all three inequalities (3.14)-(3.16).

Input : $p_1, p_2, \ldots, p_m, \delta, \epsilon, \pi = (A_0, A_{i_1}, \ldots, A_{i_m})$

1. Set $S(A_m) = 0$. Set $i = m - 1$.
2. (Schedule A_i :)

   if $i \in D$ and A_j is the downhill activity preceding A_i in π then

   $$S(A_i) = S(A_j) + (j + 1 - i)\delta + \delta + 2\epsilon$$

   if $i \in U$ and $i + 1 \in U$ then

   $$S(A_i) = S(A_{i+1}) - \delta - 2\epsilon - p_{i+1}$$

   if $i \in U$ and $i + 1 \in D$ and A_j is the uphill activity following A_i in π then

   $$S(A_i) = \min\{S(A_j) - (j - i)\delta - 2\epsilon, S(A_{i+1}) - \delta - 2\epsilon - p_{i+1}\}$$

3. If $i > 0$ set $i \rightarrow i - 1$ and goto 2, else goto 4.
4. (Compute cycle time)

$$\begin{aligned} L_1 &= S(A_{i_m}) + (i_m + 2)\delta + 2\epsilon - S(A_0). \\ L_2 &= \max_i p_i + 4(\delta + \epsilon). \\ L_3 &= \max_{i \in U, i-1 \in D} S(A_{i-1}) + \delta + 2\epsilon + p_i - S(A_i) \\ L_S &= \max\{L_1, L_2, L_3\}. \end{aligned}$$

Output : $\{S, L_S\}$

Figure 3.5: Algorithm for computing the cycle time of a pyramidal permutation

The algorithm can easily be implemented in $O(m)$ time. We now establish its correctness.

Theorem 3.4 For every pyramidal permutation π, the schedule defined by the algorithm in Figure 3.5 is feasible and has minimum cycle time among all schedules for π.

Proof. Feasibility of the schedule (3.7) can be checked by induction on i and t. In particular, for all $t \in \mathbb{N}$, $S(A_0, t+1)$ is feasible if $S(A_{i_m}, t)$ is feasible, because of (3.14). Moreover, if A_{i_j} starts at time $S(A_{i_j}, t)$, then the robot can reach machine $M_{i_{j+1}}$ before $S(A_{i_{j+1}}, t)$ (in time to perform $A_{i_{j+1}}$), because of (3.8), (3.10), (3.13). Finally, at time $S(A_i, t)$, machine M_i has finished processing and can be unloaded; this is true because of (3.10) if $i \in U$ and $i-1 \in U$; because of (3.16) if $i \in U$ and $i-1 \in D$; because of (3.11) if $i \in D$ and $i-1 \in U$; and because of (3.8) and (3.15) if $i \in D$ and $i-1 \in D$. Thus the schedule defined by (3.7) is feasible.

It remains to show that the schedule defined by the algorithm in Figure 3.5 has minimum cycle time among all schedules for π. The following relation (3.17) is crucial for an intuitive understanding of the algorithm : it expresses that the time elapsed between the execution of an uphill activity A_u and a downhill activity A_d, is at least as short in S as in any other schedule.

We now claim the schedule S to have the following property : for every feasible schedule T, for all $t \in \mathbb{N}$, for all $u \in U$ and for all $d \in D$ such that either $u \geq d$ or $\{u, u+1, \ldots, d-1\} \subseteq U$,

$$T(A_d, t) - T(A_u, t) \geq S(A_d) - S(A_u). \tag{3.17}$$

We prove this by backward induction on u, for each fixed value of d . The claim holds for $u = m$, as follows easily from (3.8). Now suppose that it holds for $u = j$, and let A_i be the uphill activity immediately preceding A_j in π. If $j = i+1$, then (3.17) follows from (3.9), (3.10) and the induction hypothesis. If $j > i+1$, then A_{i+1} is downhill and $S(A_i)$ is given by (3.13). Now if,

$$S(A_i) = S(A_j) - (j-i)\delta - 2\epsilon.$$

then (3.17) follows from (3.12) and the induction hypothesis. On the other hand if

$$S(A_i) = S(A_{i+1}) - \delta - 2\epsilon - p_{i+1},$$

then, in view of (3.11),

$$T(A_{i+1}, t) - T(A_i, t) \geq S(A_{i+1}) - S(A_i)$$

for all $t \in \mathbb{N}$. Furthermore, since $i+1 \in D$, equation (3.8) implies

$$T(A_d, t) - T(A_{i+1}, t) \geq S(A_d) - S(A_{i+1}),$$

and (3.17) follows from the latter two inequalities. This completes the proof of the claim.

Let now T be any feasible schedule for π. Letting $u = 0$ and $d = i_m$ in the claim, we have in particular

$$T(A_{i_m}, t) - T(A_0, t) \geq S(A_{i_m}) - S(A_0)$$

for all $t \in \mathbb{N}$. Therefore,

$$\begin{aligned} T(A_0, t+1) - T(A_0, t) &\geq T(A_{i_m}, t) + (i_m + 2)\delta + 2\epsilon - T(A_0, t) \\ &\geq S(A_{i_m}) + (i_m + 2)\delta + 2\epsilon - S(A_0) \\ &= L_1. \end{aligned} \tag{3.18}$$

Next, consider an index $i \in U$ such that $i - 1 \in D$ and

$$L_3 = S(A_{i-1}) + \delta + 2\epsilon + p_i - S(A_i).$$

Letting $u = i$ and $d = i - 1$ in the claim, we obtain for all $t \in \mathbb{N}$:

$$T(A_{i-1}, t) - T(A_i, t) \geq S(A_{i-1}) - S(A_i).$$

Since T is feasible, the same reasoning that led to (3.16) also establishes that

$$T(A_i, t+1) \geq T(A_{i-1}, t) + \delta + 2\epsilon + p_i.$$

The previous inequalities together imply :

$$T(A_i, t+1) - T(A_i, t) \geq L_3. \tag{3.19}$$

From (3.18), (3.19) and Lemma 3.1, we now conclude that the long run average cycle time of T is at least $L_S = \max\{L_1, L_2, L_3\}$. This completes the proof of Theorem 3.4. ■

3.5 Polynomial algorithms for the identical parts cyclic scheduling problem

Theorem 3.3 and Theorem 3.4 together imply that, for fixed m, the identical parts cyclic scheduling problem can be solved in constant time by enumerating all pyramidal permutations and subsequently computing their cycle time. However, since there are 2^{m-1} pyramidal permutations, the resulting algorithm has exponential complexity when m is considered to be part of the input. In this section, we will present more efficient algorithms, whose complexity grows only polynomially with m.

In the framework of the Traveling Salesman problem, a pyramidal tour of minimum length can be found by dynamic programming in $O(n^2)$ time, where n denotes the number of cities (see e.g. Gilmore et al.[1985]). In terms of the identical parts cyclic scheduling problem, a shortest Hamiltonion tour would correspond to a permutation with minimum cycle time. Similarly, a shortest Hamiltonian path would correspond

to a schedule in which $S(d) - S(0)$ is minimum, where d is the downhill activity with minimum index, i.e. the last activity in the permutation.

The first difficulty here stems from the fact that, in the Traveling Salesman problem, the distance between two cities is given explicitly in the distance matrix whereas in the identical parts cyclic scheduling problem, the 'distance' $S(A_{i_j}) - S(A_{i_{j+1}})$ between two consecutive activities is not a priori known, since the waiting time of the robot depends on the permutation. For the type of schedules constructed by the algorithm in the previous section, however, we will be able to show that these distances can somehow be computed online.

In this section, we first give a dynamic programming algorithm for the identical parts cyclic scheduling problem which computes, for every possible value of d, a pyramidal schedule S such that $S(d) - S(0)$ is minimum over all pyramidal schedules in which A_d is the downhill activity with minimum index. This dynamic programming algorithm is similar to the one computing a shortest *path* for the traveling salesman problem, but it does not necessarily output an optimal schedule (i.e. a *tour*) for the identical parts cyclic scheduling problem. This is the second difficulty encountered in our problem, in comparison with the traveling salesman problem. However, we show that, based on the dynamic programming formulation, an optimal schedule can be obtained in polynomial time.

Define now the following sets of permutations.

Definition 3.12 For all $u \in \{0, \ldots, m\}$ and $d \in \{1, \ldots, m\}$ with $u \neq d$, $\Pi_{u,d}$ is the set of pyramidal permutations such that:

1. A_u is uphill
2. A_d is downhill
3. if $u < d$, then A_i is uphill for all $i \in \{u, u+1, \ldots, d-1\}$
4. if $d < u$, then A_i is downhill for all $i \in \{d, d+1, \ldots, u-1\}$.

For the sake of simplicity, when $S(A_i, t)$ is a 1-periodic schedule, we use the shorthand $S(A_i)$ instead of $S(A_i, 0)$ $(i = 1, \ldots, m)$.

Now we define a function $L(u, d)$ by:

Definition 3.13 For all $u \in \{0, \ldots, m\}$ and $d \in \{1, \ldots, m\}$ with $u \neq d$,

$$L(u,d) = \min\{S_\pi(A_d) - S_\pi(A_u) | \pi \in \Pi_{u,d} \text{ and } S_\pi \text{ is a 1-periodic schedule for } \pi\}$$

Theorem 3.5 For all $u \in \{0, \ldots, m\}$ and $d \in \{1, \ldots, m\}$ with $u \neq d$, the value of $L(u, d)$ can be computed in $O(m^2)$ time by the following dynamic programming formulation:

$$L(m-1, m) = \delta + 2\epsilon + p_m,$$

$$L(m, m-1) = 2\epsilon + 3\delta$$

and, for all $\{u,d\} \neq \{m-1,m\}$,

$$L(u,d) = \begin{cases} L(u,d+1) + 3\delta + 2\epsilon & \text{if } u > d+1 \\ \min_{j>u}\{L(u,j) + (j-d+2)\delta + 2\epsilon\} & \text{if } u = d+1 \\ L(u+1,d) + \delta + 2\epsilon + p_{u+1} & \text{if } u < d-1 \\ \min_{j>d}\{\max\{L(j,d) + 2\epsilon + (j-u)\delta, \delta + 2\epsilon + p_d\}\} & \text{if } u = d-1 \end{cases}$$

Proof. The expressions for $L(m-1,m)$ and $L(m,m-1)$ are easily checked to be correct (see (3.8) and (3.10)). For all other values of (u,d), the recursive equations are based on the algorithm given in the previous section (Figure 3.5). Their validity can be checked by induction. For example, assume that the value of $L(u,j)$ is correctly computed by these equations for all $j > u$, and consider next $L(u,u-1)$ (i.e., $u = d+1$). We must find a pyramidal permutation π and a corresponding schedule S which minimizes $S(A_{u-1}) - S(A_u)$. For any given permutation π, let A_j be the downhill activity immediately preceding A_{u-1} in π. From equation (3.8), we know that

$$S(A_{u-1}) = S(A_j) + (j-u+3)\delta + 2\epsilon.$$

Moreover, relying on the dynamic programming principle of optimality, we can assume that $S(A_j) - S(A_u)$ is as small as possible under the previous restrictions, i.e. $S(A_j) - S(A_u) = L(u,j)$. It follows now that

$$\begin{aligned} S(A_{u-1}) - S(A_u) &= L(u,j) + (j-u+3)\delta + 2\epsilon \\ &= L(u,j) + (j-d+2)\delta + 2\epsilon. \end{aligned}$$

Thus, $L(u,u-1)$ is attained by a permutation π which minimizes the previous expression, as is asserted in the statement of the theorem. The other cases are left to the reader.

As for the complexity of the formulation, notice that the value of each $L(u,d)$ with $|u-d| \geq 2$ can be computed in constant time. The computation of each $L(u,d)$ with $|u-d| = 1$ requires $O(m)$ time, but there are only $2m$ pairs (u,d) such that $|u-d| = 1$. Thus, all values $L(u,d)$ can be obtained in $O(m^2)$ time. ∎

The dynamic programming formulation in Theorem 3.5 allows to compute in $O(m^2)$ time, for every possible last activity $A_d, d = 1,\ldots,m$:

- the value of $L(0,d)$
- a permutation $\pi_d \in \Pi_{0,d}$.
- a schedule S_{π_d} for π_d such that $S_{\pi_d}(A_d) - S_{\pi_d}(A_0) = L(0,d)$.

The schedule S_{π_d} is the same schedule that would have been output by the algorithm in Figure 3.5, had it taken π_d as input. It follows then that the cycle time of the permutation π_d produced by the dynamic programming algorithm can be computed as in step 4 of the algorithm in Figure 3.5. But again we emphasize here that the permutation π_d output by the dynamic programming algorithm does not necessarily have minimum cycle time. In the remainder of this section, we explain how the dynamic programming formulation can be used to solve the identical parts cyclic scheduling problem to optimality.

Let us first focus on the recognition version of the problem, which may be stated as :

INPUT : $p_i, i = 1, \ldots, m, \delta, \epsilon, C$

QUESTION : Is there a 1-periodic schedule with cycle time at most C ?

This problem can be solved by a slight adaptation of the dynamic programming algorithm. Informally, the dynamic programming algorithm will be modified so that, when it finds a permutation, then the cycle time of the permutation is less than or equal to C, and when it does not find a permutation, then such a permutation does not exist.

To start with, let us assume from now on that $\max_i p_i + 4(\delta+\epsilon) \leq C$, since otherwise Theorem 3.2 provides a negative answer to the recognition problem. Consider next the following definition, motivated by the computation of the bound L_3 in Figure 3.5 :

Definition 3.14 For all $u \in \{0, \ldots, m\}$ and $d \in \{1, \ldots, m\}$ with $u \neq d$,

$$L_C(u,d) = \min S_\pi(A_d) - S_\pi(A_u)$$

$$\text{s.t.} \quad \pi \in \Pi_{u,d}$$

$$S_\pi \text{ is a 1-periodic schedule for } \pi$$

$$\max_{i \in U, i \geq u, i-1 \in D, i-1 \geq d} \{S_\pi(A_{i-1}) + \delta + 2\epsilon + p_i - S_\pi(A_i)\} \leq C.$$

We let $L_C(u,d) = +\infty$ if there is no feasible solution to the optimization problem in Definition 3.14. Notice that $L_C(0,d) < +\infty$ for all $d \in \{1, \ldots, m\}$, since the permutation $(A_0, A_1, \ldots, A_{d-1}, A_m, A_{m-1}, \ldots, A_d)$ admits a schedule which satisfies all constraints in the definition of $L_C(0,d)$. It can be checked as in Theorem 3.5 that the values $L_C(u,d)$ can be computed in $O(m^2)$ time by the following recursion (where, for the sake of compactness, we denote by $K(u,d)$ the quantity $\min_{j>u}\{L_C(u,j) + (j-d+2)\delta + 2\epsilon\}$) :

$$L_C(m-1,m) = \delta + 2\epsilon + p_m,$$

$$L_C(m,m-1) = 2\epsilon + 3\delta$$

and, for all $\{u,d\} \neq \{m-1,m\}$,

$$L_C(u,d) = \begin{cases} L_C(u,d+1) + 3\delta + 2\epsilon & \text{if } u > d+1 \\ K(u,d) & \text{if } u = d+1 \text{ and } K(u,d) + \delta + 2\epsilon + p_u \leq C \\ +\infty & \text{if } u = d+1 \text{ and } K(u,d) + \delta + 2\epsilon + p_u > C \\ L_C(u+1,d) + \delta + 2\epsilon + p_{u+1} & \text{if } u < d-1 \\ \min_{j>d}\{\max\{L_C(j,d) + 2\epsilon + (j-u)\delta, \delta + 2\epsilon + p_d\}\} & \text{if } u = d-1 \end{cases}$$

Theorem 3.6 The recognition version of the identical parts cyclic scheduling problem can be solved in $O(m^2)$ time.

Proof. We can compute in $O(m^2)$, for each $d \in \{1,\ldots,m\}$:

- the value of $L_C(0,d)$
- a permutation $\pi_d \in \Pi_{0,d}$.
- a schedule S_{π_d} for π_d such that $S_{\pi_d}(A_d) - S_{\pi_d}(A_0) = L(0,d)$ and S_{π_d} satisfies the third constraint in Definition 3.14.

We claim that the answer to the recognition problem is affirmative if and only if there exists $d \in \{1,\ldots,m\}$ such that

$$L_C(0,d) + (d+2)\delta + 2\epsilon \leq C. \tag{3.20}$$

Indeed, if (3.20) holds for some d, then the cycle time of π_d is at most C (see Step 4 in Figure 3.5), and we are done. Conversely, assume that there exists a pyramidal permutation, say π, whose cycle time is at most C. Let A_d be the last downhill activity in π, and let S_π be the schedule computed for π by the algorithm in Figure 3.5. By Definition 3.14, $L_C(0,d) \leq S_\pi(A_d) - S_\pi(A_0)$. Moreover, in view of step 4 in Figure 3.5, $S_\pi(A_d) - S_\pi(A_0) + (d+2)\delta + 2\epsilon \leq C$. Thus we conclude that (3.20) holds, which concludes the proof. ∎

Of course, the optimization version of the problem can be solved by repeatedly solving the recognition version, while applying binary search between the lowerbound and the upperbound given in Theorem 3.2 :

Corollary 3.1 For integral values of $p_i, i = 1,\ldots,m$, δ, ϵ the optimization version of the identical parts m -machine cyclic scheduling problem can be solved in $O(m^2 \log(m\delta))$ time.

In the last part of this section, we now describe a strongly polynomial algorithm to solve the optimization version of the identical parts m -machine cyclic scheduling problem. We first need yet another modification of Definition 3.13, in which some activities are 'forced' to be downhill (the motivation for this definition should become clear very shortly).

Definition 3.15 For all $F \subseteq \{1,\ldots,m\}, u \in \{0,\ldots,m\} \setminus F$ and $d \in \{1,\ldots,m\}$ with $u \neq d$,

$$\begin{aligned} L_F(u,d) \quad = \quad & \min S_\pi(A_d) - S_\pi(A_u) \\ \text{s.t.} \quad & \pi \in \Pi_{u,d} \\ & S_\pi \text{ is a 1-periodic schedule for } \pi \\ & A_i \text{ is downhill in } \pi, \text{ for all } i \in F. \end{aligned}$$

For any $F \subseteq \{1,\ldots,m\}$, a straightforward adaption of our previous dynamic programming algorithm, which simply 'skips' all pairs (u,d) such that $u \in F$, allows to compute in $O(m^2)$ time, for each $d \in \{1,\ldots,m\}$:

- the value of $L_F(0,d)$
- a permutation $\pi_{F,d}$ such that A_i is downhill in $\pi_{F,d}$ for all $i \in F$
- a schedule $S_{F,d}$ for $\pi_{F,d}$ such that $S_{F,d}(A_d) - S_{F,d}(A_0) = L_F(0,d)$.

The cycle time of $\pi_{F,d}$ (as computed by the algorithm in Figure 3.5) is denoted by $L(\pi_{F,d})$. Suppose now that $L(\pi_{F,d}) = L_3$. We call activity A_i an *obstruction* of $\pi_{F,d}$ if A_i is uphill in $\pi_{F,d}$, A_{i-1} is downhill in $\pi_{F,d}$ and $L(\pi_{F,d}) = S_{F,d}(A_{i-1}) + \delta + 2\epsilon + p_i - S_{F,d}(A_i)$. Roughly speaking, the intuition behind the algorithm that we are about to present is that, if a current schedule is not optimal, then it must contain an obstruction A_i, and A_i should be downhill in any optimal schedule. This property is stated more precisely in the following two lemmas.

Lemma 3.3 For all $F \subseteq \{1,\ldots,m\}$ and $d \in \{1,\ldots,m\}$, if there is no obstruction in $\pi_{F,d}$, then there is no pyramidal permutation with cycle time less than $L(\pi_{F,d})$ in which all activities in F are downhill and A_d is the last downhill activity.

Proof. If $L(\pi_{F,d}) = \max_{1 \leq i \leq m} p_i + 4(\delta + \epsilon)$, then $\pi_{F,d}$ is optimal by Theorem 3.2. If this is not the case, and there is no obstruction in $\pi_{F,d}$, then by definition of the bound L_1 in Figure 3.5

$$T(A_0, t+1) - T(A_d, t) \geq S_{F,d}(A_0, t+1) - S_{F,d}(A_d, t)$$

for every feasible schedule T. Combined with the definition of $L_F(0,d)$, this proves the lemma. ∎

Lemma 3.4 For all $F \subseteq \{1,\ldots,m\}$ and $d \in \{1,\ldots,m\}$, if A_i is any obstruction in $\pi_{F,d}$, then there is no pyramidal permutation with cycle time less than $L(\pi_{F,d})$ in which all activities in F are downhill and A_i is uphill.

Proof. Let π be any pyramidal permutation in which all activities in F are downhill and A_i is uphill, and let T be a shortest 1-periodic schedule for π. Suppose first that A_{i-1} is downhill in π. Notice that, by definition,

$$L_F(i,i-1) = S_{F,d}(A_{i-1}) - S_{F,d}(A_i) \leq T(A_{i-1},t) - T(A_i,t).$$

On the other hand, $\delta + 2\epsilon + p_i$ is a lowerbound on $T(A_i,t+1) - T(A_i,t)$. Thus

$$\begin{aligned} L(\pi_{F,d}) &= S_{F,d}(A_{i-1}) + \delta + 2\epsilon + p_i - S_{F,d}(A_i) \\ &\leq T(A_i,t+1) - T(A_i,t) \end{aligned}$$

as required.

Next, suppose that both A_i and A_{i-1} are uphill in π. Then,

$$\begin{aligned} T(A_i,t+1) - T(A_{i-1},t+1) &\geq \delta + 2\epsilon + p_i \\ &= S_{F,d}(A_i,t+1) - S_{F,d}(A_{i-1},t). \end{aligned}$$

Now consider the first point after $T(A_i,t)$, say τ, at which the robot reaches M_{i-1} after travelling trajectory (M_i,M_{i-1}). By definition of $L_F(i,i-1)$,

$$\tau - T(A_i,t) \geq S_{F,d}(A_{i-1},t) - S_{F,d}(A_i,t)$$

because the permutation π', obtained by switching the status of A_{i-1} from uphill to downhill in π, admits a shortest schedule T' such that $T'(A_i,t) = T(A_i,t)$ and $T'(A_{i-1},t) = \tau$, as implied by the algorithm in Figure 3.5. Combining the latter inequalities one derives that

$$\begin{aligned} L(\pi) &= T(A_i,t+1) - T(A_i,t) \\ &\geq T(A_i,t+1) - T(A_{i-1},t+1) + \tau - T(A_i,t) \\ &\geq S_{F,d}(A_i,t+1) - S_{F,d}(A_{i-1},t) + S_{F,d}(A_{i-1},t) - S_{F,d}(A_i,t) \\ &= L(\pi_{F,d}) \end{aligned}$$

as required. ∎

Combining Lemma 3.3. and Lemma 3.4. we obtain the following result :

Theorem 3.7 The optimization version of the identical parts m-machine cyclic scheduling problem can be solved in $O(m^3)$ time.

Proof. We claim that the algorithm in Figure 3.6 correctly solves the problem:

Input : $p_1, p_2, \ldots, p_m, \delta, \epsilon$

1. $F \rightarrow \emptyset$, $opt \rightarrow +\infty$
2. Call the dynamic programming algorithm and compute $L(\pi_{F,d}), \pi_{F,d}, S_{F,d}$ for $d = 1, \ldots, m$.
3. Select d such that

$$S_{F,d}(A_d) - S_{F,d}(A_0) + (d+2)\delta + 2\epsilon =$$

$$\min_j \{S_{F,j}(A_j) - S_{F,j}(A_0) + (j+2)\delta + 2\epsilon\}$$

$$opt \rightarrow \min(opt, L(\pi_{F,d})).$$

4. If $opt = \max_i p_i + 4(\delta + \epsilon)$, return $\pi_{F,d}$ and stop.
5. (Lemma 3.3.) If there are no obstructions in $\pi_{F,d}$, return a permutation with cycle time equal to opt, and stop.
6. (Lemma 3.4.) Let I be the set of obstructions in $\pi_{F,d}$. Set $F \rightarrow F \cup I$ and goto 2.

Output : (optimal permutation and its cycle time) $\{\pi^*, L_{\pi^*}\}$

Figure 3.6: Algorithm for computing a permutation with minimum cycle time and its cycle time.

Observe that the complexity of the algorithm in Figure 3.6 is indeed $O(m^3)$ since $|F| \leq m$, and hence the dynamic programming algorithm cannot be called more than m times.

To see that the algorithm is correct, assume first that the following property (P) holds before some iteration of Step 2 : (P) if there is a permutation, say π, with cycle time smaller than opt, then all activities in F are downhill in π (notice that property (P) certainly holds before the first iteration of Step 2.) Under this assumption, we are going to prove that property (P) is an invariant of the algorithm, i.e. : if property (P) holds before some iteration of Step 2, then either the algorithm returns an optimal value in the subsequent execution of Steps 4-5, or property (P) holds again before the next iteration of Step 2.

Indeed, if the algorithm stops in Step 4, then $\pi_{F,d}$ is optimal by Lemma 3.1. Suppose now that it stops in Step 5, and (by contradiction) that there exists a permutation π with cycle time $L(\pi) < opt$. Let S_π be any schedule for π, and let A_j be the last downhill activity in π. By the property (P), all activities in F are downhill in π. Hence by Definition 3.15 and by definition of $S_{F,j}$:

$$S_\pi(A_j) - S_\pi(A_0) \geq S_{F,j}(A_j) - S_{F,j}(A_0).$$

Moreover,

$$L(\pi) \geq S_\pi(A_j) - S_\pi(A_0) + (j+2)\delta + 2\epsilon,$$

and thus

$$L(\pi) \geq S_{F,j}(A_j) - S_{F,j}(A_0) + (j+2)\delta + 2\epsilon.$$

Step 3 of the algorithm implies now:

$$L(\pi) \geq S_{F,d}(A_j) - S_{F,d}(A_0) + (d+2)\delta + 2\epsilon.$$

Since $\pi_{F,d}$ has no obstruction, the previous inequality boils down to

$$L(\pi) \geq L(\pi_{F,d}),$$

which contradicts $L(\pi) < opt$.

Finally if the algorithm does not stop in either Step 4 or 5, then $\pi_{F,d}$ must contain a set of obstructions, denoted by I. Setting $F \rightarrow F \cup I$, property (P) is now directly implied by Lemma 3.4

Thus property (P) is indeed an invariant of the algorithm, and we conclude that the algorithm is correct (since it is finite). ∎

3.6 Summary and directions for further research

Planning and scheduling in modern production environments, such as robotic cells, gives rise to a variety of challenging decision problems that do not fit well into classical models. In this chapter, we have studied a throughput rate maximization problem in a flowshop-like robotic cell in which the material handling system consists of a single robot or robot arm. The throughput rate of the cell is highly dependent on the interaction between the material handling system and the machines processing the parts. We have shown that, when there is only one type of parts to be produced, the problem can be solved in (strongly) polynomial time even if the number of machines is viewed as an input parameter of the problem. This generalizes previous results established by Sethi et al. [1992] for the three-machine case. Interestingly, our analysis makes heavy use of seemingly unrelated concepts and techniques investigated by various

authors in connection with the traveling salesman problem (although, it should be observed, we never actually obtain a TSP-formulation of our problem).

Many interesting related problems are still open. The first open problem we mention is the conjecture of Sethi et al. [1992], which is to be addressed in the next Chapter, that 1-unit cycles are optimal among all possible robot move sequences, in the case where there is only one part-type to be produced. Other interesting open problems concern the case where there is more than one part-type. The applicability of the concept of pyramidal permutations to such situations seems to be limited for a number of reasons. First of all, there exist problem instances with multiple part types in which 1-unit cycles can be shown to be dominated (see Hall et al. [1995a]). Second, even if we restrict the analysis to 1-unit cycles, it is not clear whether there always exists an optimal permutation that is pyramidal. Finally, an NP-hardness result of Hall et al. [1995b] (mentioned in the introduction) establishes that computing the optimal part input sequence in a three machine robotic cell is NP-hard for the downhill permutation, and thus for pyramidal permutations in general. We also notice that the complexity of the multiple parts problem remains open if either the number of parts or the number of part-types is fixed. This question is briefly addressed in Hall et al. [1995b]. As a matter of fact, to the best of our knowledge, the question appears to be open even for ordinary three machine flowshops (without robot). Related issues have been recently investigated by Agnetis [1989], Hochbaum & Shamir[1991], Granot et al.[1993], etc.

Chapter 4

The optimality of short robot move sequences in a robotic flowshop

4.1 Introduction

Many authors have investigated robot move sequencing problems in robotic cells. Robot move sequencing problems ask to find a robot move sequence that yields an optimal throughput rate or cycle time. Often, such sequencing problems turn out to be hard to solve, or even analyse. The analysis becomes easier however, when it is restricted to the subset of sequences that are obtained by repeating so called 1-unit cycles (1-unit cycles are defined in Chapters 2 and 3). This restriction is common practice, although for most models it is known that the restriction may cause an increase in the smallest attainable cycle time. As an exception, Sethi et al. [1992] conjecture that in three machine robotic cells of the type studied in Chapter 3, the smallest attainable cycle time can always be attained by a 1-unit cycle. Sriskandarajah et al. [1995] show that 1-unit cycles dominate certain classes of 2 and 3-unit cycles. The present chapter also addresses the conjecture. We show that no finite length robot move sequence can have smaller cycle time than the smallest cycle time attainable by a 1-unit cycle.

Some of the notations and definitions we use in this chapter borrow from Chapters 2 and 3. For a thorough introduction to this chapter the reader is refered to Sections 3.1 and 3.2, and Section 3.3 upto and including Theorem 3.1. Section 4.2 introduces a state space model to analyse a three machine robotic cell of the type described in Chapter 2. Section 4.3 gives the results. Section 4.4 discusses directions for further research.

4.2 A state space graph model

We are interested in the behavior of the cell when producing an infinite batch of parts. More precisely, let S be a schedule for the robot (schedules are defined in Definition

3.3). We are interested in schedules minimizing

$$\lim_{t\to\infty} \frac{S(A_m, t)}{t}.$$

Sethi et al. conjecture that this minimum can be attained by a 1-unit cycle. We prove that the conjecture holds when the minimum is taken over the set of schedules that can be obtained by repeating infinitely many times a robot move sequence of finite length.

Our proof is based on a state space graph model for the cell which is explained in this section. The state of a three machine robotic cell is completely specified, when the following information is available :

1. For each machine whether it is loaded or not.
2. For each machine, the remaining processing time of the part present on the machine, if there is one.
3. The state of the robot, i.e. the position of the robot, and whether it is carrying a part or not.

In this chapter we distinguish between *cell state* and *loaded/unloaded state*. A loaded/unloaded state only specifies for each machine whether it is loaded or not. This chapter presents a graph model in which there are two set of vertices associated with each possible loaded/unloaded state of the cell. Each loaded/unloaded state can be represented by a 3-dimensional $(0,1)$-vector v, where $v_i = 1$ indicates that machine M_i is loaded, and $v_i = 0$ indicates that it is unloaded.

The vertices in the graph correspond to cell states. A cell state specifies not only the loaded/unloaded state, but also the remaining processing times on the machines and the robot position. We only consider cell states occuring when the robot is not carrying a part. (In such a situation, each part in the cell is on one of the machines.) Consequently, we also only consider state transitions between cell states occurring when the robot is not carrying a part. Thus, when going from a cell state with loaded/unloaded vector $(1,1,0)$ to a cell state with loaded/unloaded vector $(1,0,1)$ we need not go via $(1,0,0)$. All states of the cell occuring when the robot is not carrying a part can be completely specified by a septuple $(v_1, v_2, v_3, r_1, r_2, r_3, p)$ where r_i denotes the remaining processing time on machine M_i and p denotes the robot position. With each loaded/unloaded state v, we associate two sets of vertices. One set, denoted $e(v)$, contains all states of the cell where the loaded/unloaded state has just become as specified by v. The other set, denoted $l(v)$, contains all states of the cell where it is just about to leave v. Formally :

$$e(v) = \{(v, r, M_i) : v_i = 1, r_i = p_i, v_{i-1} = 0\} \cup \{(v, r, M_4) : v_3 = 0\},$$

$$l(v) = \{(v, r, M_i) : v_i = 1, r_i = 0, v_{i+1} = 0\} \cup \{(v, r, M_0) : v_1 = 0\}.$$

We need not specify whether the robot is carrying a part or not since it must be the case that, if we have just entered a cell state, the robot has just been unloaded,

and if we are just about to leave a cell state, the robot is unloaded because it is just about to unload a machine. Feasibility of the septuples is defined with respect to both the loaded/unloaded state v, and whether we are considering a node in $e(v)$ or in $l(v)$. First notice that $v_i = 0$ implies that $r_i = 0$. Second, suppose that just after performing activity A_i we have entered some cell state S, then the robot is at position M_{i+1} in S. Similarly, if we are just about to leave some cell state S by performing A_i, then the robot must be at M_i in S.

We draw two types of arcs in the graph. First, for each v, we draw an arc of length t from a cell state S in $e(v)$ to a cell state S' in $l(v)$ if, when in cell state S, it is possible to reach S' in exactly t time units. Second, we draw an arc of length δ from a cell state S in $l(v)$ to a cell state S' in $e(w)$ if there is a single activity that yields the corresponding transition in exactly δ times units. (See Figure 4.1.)

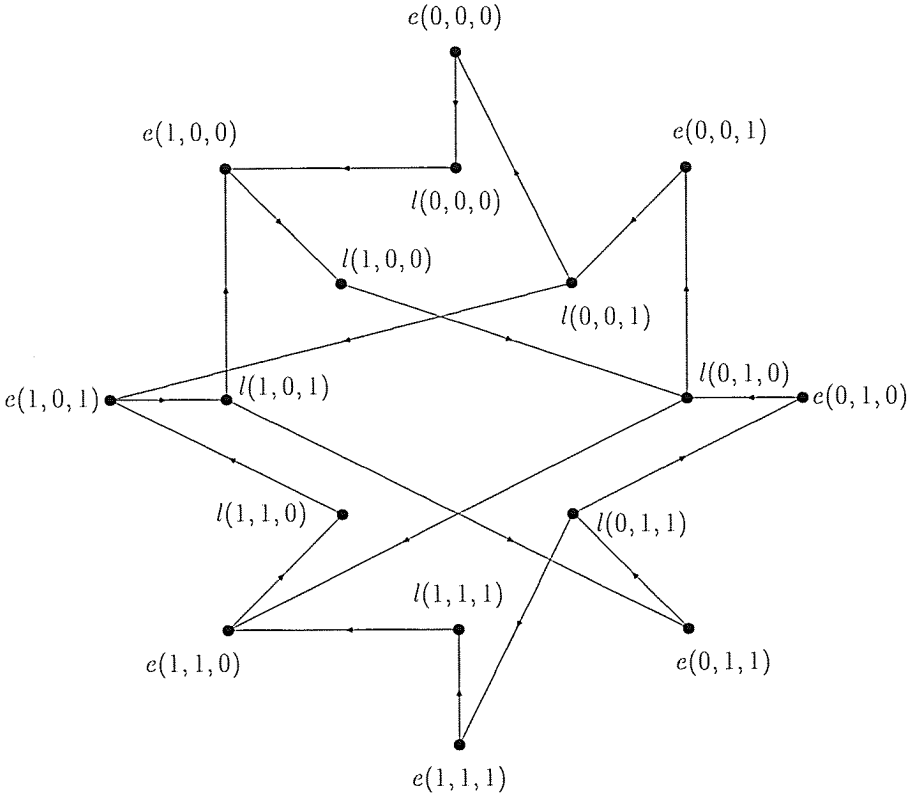

Figure 4.1: The state space graph

We are going to examine this graph and derive properties that enable us to show that a slightly weaker version of the conjecture of Sethi et al. [1992] holds. We only consider the case where $\delta_i = \delta$ and $\epsilon_i = 0$ for all i (thus there is no loading and unloading time). In this case, we know from Lemma 3.2 that if $\max_i p_i \geq 8\delta$, the downhill sequence has minimum long run cycle time. Lemma 3.1 then establishes the optimality of the downhill permutation over all possible robot move sequences, whether they are finite or not. Hence we only need to prove the conjecture for the case where $\max_i p_i < 8\delta$. In that case, the downhill sequence achieves a cycle time of 12δ, consisting of travel time only. We conclude that we only need to prove the conjecture for cases where the optimal cycle time does not exceed 12δ.

Now, let us consider the set $e(0,0,1)$. In order to have entered a cell state in this set, the last executed activity must be A_2, and consequently the robot is at M_3. Moreover the remaining processing times are $(0,0,p_3)$. Thus, $|e(0,0,1)| = 1$. Consider also $l(1,0,0)$. To leave a cell state in this set, the robot must perform A_1, hence it must be at M_1, and the remaining processing times are $(0,0,0)$. Again we have $|l(1,0,0)| = 1$. There is a certain symmetry in these two cases. If the parts were to flow through the cell in the opposite direction, $l(0,0,1)$ would take the role of $e(1,0,0)$ and vice versa. We do not exploit this symmetry in the remainder of the analysis (although we think observations of this type could be helpful, e.g. when proving the optimality of 1-unit cycles in an m-machine robotic cell).

For a finite robot move sequence, say π, consider the following two cases. The first case is where in the course of executing π, the cell reaches at least once the single cell state in the set $e(0,0,1)$ or the single cell state in the set $l(1,0,0)$. Suppose it reaches $e(0,0,1)$, and this occurs exactly once (the other subcases are similar). Let n be the number of parts delivered when π is executed once. Furthermore, we define an *active schedule* as the set of starting times that are realised when the robot executes each move as quickly as possible. Let the *execution time* t be the amount of time it takes the robot to execute an active schedule for π once, i.e. the amount of time that elapses between two occurences of $e(0,0,1)$ in an active schedule for π.

It can be seen as follows that no other schedule T for π takes time less than t. The execution time is the sum of the robot travel time and the robot waiting time. Since the robot travel time depends on π and not on the schedule, T can only improve on the active schedule by reducing waiting time. Now, for T to induce less waiting time than the active schedule, there must be some first robot move that is performed earlier in T than it is performed in the active schedule. To be performed earlier than in the active schedule, the sum of travel and waiting time in the active schedule up to this activity must be larger. By definition, waiting arises in the active schedule only if the robot must unload a machine that has not finished processing yet. Since we are considering a first robot move that is performed earlier than in the active schedule, we know that the active schedule has waiting time before executing this robot move, and hence it is an unloading operation. On the other hand, since we are considering the first robot move that is executed earlier in T than in the active schedule, the loading of this machine in the active schedule has not taken place later than in T. Thus, it cannot be unloaded

in T before it is unloaded in the active schedule. We arrive at a contradiction and conclude that the active schedule has shortest possible execution time. It follows from the above discussion that the cycle time of π equals $\frac{t}{n}$ and that this cycle time can easily be attained.

The second case is where neither the single cell state in the set $e(0,0,1)$ nor the single cell state in the set $l(1,0,0)$ occur while executing π. We shall see in the remainder that this implies that the cycle time of an optimal sequence equals 12δ.

The notion of active schedules is not only useful in the previous discussion. In fact, the same reasoning leads to the conclusion that there always is an optimal schedule, i.e. a schedule with minimum long run cycle time, among the active schedules.

Consider some robot move sequence π. Clearly, a schedule for such a sequence induces a sequence of cell states, say σ_π. Similarly, any sequence σ of cell states that results from an edge traversal in the graph defined above, induces a robot move sequence, say π_σ. Further, it is not hard to see that $\pi_{\sigma_\pi} = \pi$. We conclude that instead of schedules for robot move sequences, we can restate our problem in terms of edge traversals. In the remainder we analyse properties of such edge traversals.

Suppose that in an active schedule for some optimal edge traversal σ^*, the cell is more than once in the unique cell state in the set $e(0,0,1)$ in the course of executing σ^*. Let d be the time elapsed between two occurrences of this state, and let l be the number of parts delivered at the output station during this time interval. Then, since π is finite, it must hold that $\frac{d}{l}$ equals the minimum long run average cycle time. Hence, should it happen that $e(0,0,1)$ occurs twice in some optimal sequence, then we may assume that repeating the sequence of activities that are performed between these two occurences is also optimal. Exactly the same reasoning holds of course for $l(1,0,0)$. Thus, we distinguish the following four cases:

1. Neither $l(1,0,0)$ nor $e(0,0,1)$ occurs in an optimal sequence.

2. $l(1,0,0)$ occurs (once) in some optimal sequence, but $e(0,0,1)$ does not.

3. $e(0,0,1)$ occurs (once) in some optimal sequence, but $l(1,0,0)$ does not.

4. both $l(1,0,0)$ and $e(0,0,1)$ occur (once) in some optimal sequence.

Since these cases are exhaustive, at least one of them applies.

If executing σ^* once causes the robot to deliver n parts at the output device, the assumption that σ^* is such that it can be repeated infinitely many times, implies that machine M_1 is unloaded n times each time σ^* is executed. This means that the edge traversal contains n edges that lead from a cell state in a set $l(v)$ such that $v_1 = 1$ and $v_2 = 0$, to a cell state in a set $e(w)$ such that $w_1 = 0$ and $w_2 = 1$.

In the remainder we use some shorthand notation. For example we use 101 to refer to a loaded/unloaded state v, having $v_1 = 1, v_2 = 0, v_3 = 1$. Further, for a sequence of succesive cell states in the sets $e(1,0,1)$, $l(1,0,1)$, $e(0,1,1)$, and $l(0,1,1)$, we use the shorthand 101 011. We use '/' as logical or, i.e. 111/010 means loaded/unloaded

state 111 *or* loaded/unloaded state 010. By (101 011 111 110)^l, we mean a sequence consisting of zero (if $l = 0$) or more (e.g. one if $l = 1$) repetitions of the sequence 101 011 111 110.

4.3 Results

In this section we show that for each of the four cases distinguished above, there is a 1-unit cycle that has minimum long run average cycle time. From Chapter 3 we know that there are four pyramidal permutations, each of which can be uniquely optimal. We will see in this section that there is a one to one relation between the distinguished cases and pyramidal permutations.

Lemma 4.1 If Case 1 applies, the downhill sequence 101 011 111 110 achieves optimal cycle time.

Proof. If case 1 applies, some optimal sequence σ^* is such that the cell never enters states in which the loaded/unloaded state is $(1,0,0)$ or $(0,0,1)$. We let n again be the number of parts delivered at the output device in a single execution of σ^*. In the course of executing σ^*, a cell state in the set $l(1,0,1)$ is reached n times. Since the cell may not reach a cell state in which the loaded/unloaded state is $(1,0,0)$ or $(0,0,1)$ while proceeding from a cell state in the set $l(1,0,1)$ to the next cell state in this set, this only leaves the following two possibilities for edge traversals between these two cell states (as can be checked from Figure 4.1):

101 011 111 110

101 011 010 110

Observe that the first sequence generates the downhill permutation. Now, notice that $e(1,0,1)$ is in both sequences reached via $l(1,1,0)$, and thus is reached via the same robot activity in both sequences (the robot travels with a part from M_2 to M_3). We now show that, regardless of the r_i, after reaching a cell state in the set $l(1,1,0)$, the downhill sequence 101 011 111 110 (refered to as Case 1.a) always is preferable to the sequence 101 011 010 110 (Case 1.b). To do so, we show that, the total time elapsed between two occurences of $l(1,1,0)$ is never smaller for the latter sequence, and that if, for a given begin cell state, r_i, $i = 1,2,3$ are the remaining processing times after execution of the first sequence, and s_i, $i = 1,2,3$ are the remaining processing times after execution of the second sequence, then $r_i \leq s_i$ for $i = 1,2,3$.

Informally, this can be seen as follows. In the first alternative, the parts that are unloaded are unloaded later, and the parts that are loaded are loaded earlier. Now, since the robot travel time is the same in both sequences, we are done.

To establish this more formally, let us consider the succession of robot moves in both sequences, starting from some cell state in $l(1,1,0)$. All moves are identical up to

and including the first cell state in $e(0,1,1)$. Let $(0,p_2,\gamma)$ be the remaining processing times at this point. The possible continuations are given in Tables 4.1 and 4.2. Tables 4.1 and 4.2 simulate sequences, as do the other tabels in this Chapter. These tables are explained as follows. There is one row for each cell state reached. The first column of each table gives for each cell state, the set containing it. The second column gives, for each cell state and for each loaded machine, the required processing time minus the time elapsed since the machine has been loaded. Thus, a positive entry in this column indicates remaining processing time. If an entry is negative, then the corresponding machine has finished processing. For unloaded machines, we set the entry to zero. The third column gives the position of the robot. Finally, the fourth column gives the transition time, i.e. the amount of time elapsed between the cell state in the previous row, and the cell state in the current row.

The series of cell states of Cases 1a and 1b are then listed in Tables 4.1 and 4.2 respectively, together with the transition times between cell states (in the last line of each table, we have used our blanket assumption that $p_2 < 8\delta$).

set	remaining proc. time	r.p.	transition time
$l(1,1,0)$	immaterial	M_2	0
$e(0,1,1)$	$(0,p_2,\gamma)$	M_2	$\geq 4\delta$
$l(0,1,1)$	$(0,p_2-2\delta,\gamma-2\delta)$	M_0	2δ
$e(1,1,1)$	$(p_1,p_2-3\delta,\gamma-3\delta)$	M_1	δ
$l(1,1,1)$	$(p_1-w,p_2-3\delta-w,0)$	M_3	$\max(2\delta,\gamma-3\delta)=w$
$e(1,1,0)$	$(p_1-\delta-w,p_2-4\delta-w,0)$	M_4	δ
$l(1,1,0)$	$(p_1-3\delta-w,0,0)$	M_2	$\max(2\delta,p_2-4\delta-w)=2\delta$

Table 4.1: Case 1.a

set	remaining proc. time.	r.p.	transition time
$l(1,1,0)$	immaterial	M_2	0
$e(0,1,1)$	$(0,p_2,\gamma)$	M_2	$\geq 4\delta$
$l(0,1,1)$	$(0,p_2-w',0)$	M_3	$\max(\delta,\gamma)=w'$
$e(0,1,0)$	$(0,p_2-\delta-w',0)$	M_4	δ
$l(0,1,0)$	$(0,p_2-5\delta-w',0))$	M_0	4δ
$e(1,1,0)$	$(p_1,p_2-6\delta-w',0)$	M_1	δ
$l(1,1,0)$	$(p_1-\delta,0,0)$	M_2	$\max(\delta,p_2-6\delta-w')=\delta$

Table 4.2: Case 1.b

One easily verifies that the total transition time between the first and the second occurence of $l(1,1,0)$ is no larger in Case 1.a than in Case 1.b (since $w-2\delta \leq w'-\delta$).

Moreover, the remaining processing time on machine 1 is also no larger in Case 1.a than in Case 1.b. ■

In fact we have proved slightly more than announced: the proof of Lemma 4.1 shows that the long run cycle time of an arbitrary sequence π *cannot* increase when the subsequence (101 011 010 110) is replaced by (101 011 111 110) within π.
when starting anywhere from $l(1,1,0)$ to $e(0,1,1)$.

Observation 4.1 In Case 2, the sequence of loaded/unloaded states following $l(1,0,0)$ in the optimal sequence is of the type:

100 010 110 (101 011 111/010 110)* 101,

Lemma 4.2 In Case 2, the sequence of loaded/unloaded states following $l(1,0,0)$ in some optimal sequence is :

100 010 110 101.

Proof. First of all, it follows from the comment following Lemma 4.1 that, in the sequence displayed in Observation 4.1, only the downhill subsequence (101 011 111 110)* occurs, but not (101 011 010 110)*.

Informally, we prove the lemma by showing that, for every $l \in \mathbb{N}^+$, the cycle time of 100 010 110 (101 011 111 110)^l 101 is at least $(l+1)$ times the minimum of the cycle times of 100 010 110 101, and 101 011 111 110, the two 1-unit cycle constituents of the sequence under consideration.

Now, formally, the scenario of Table 4.3 materializes when $l = 0$. It follows from Table 4.3 that in this case the cycle time amounts to $6\delta + p_3 + x + w \geq 10\delta + p_3$, and hence, $p_3 \leq 2\delta$ (otherwise, the downhill permutation would have smaller cycle time).

set	remaining proc. time	r.p.	transition time
$l(1,0,0)$	(0,0,0)	M_1	0
$e(0,1,0)$	$(0,p_2,0)$	M_2	δ
$l(0,1,0)$	$(0,p_2-2\delta,0)$	M_0	2δ
$e(1,1,0)$	$(p_1,p_2-3\delta,0)$	M_1	δ
$l(1,1,0)$	$(p_1-x,0,0)$	M_2	$\max(\delta,p_2-3\delta)=x$
$e(1,0,1)$	$(p_1-x-\delta,0,p_3)$	M_3	δ
$l(1,0,1)$	$(p_1-p_3-x-\delta,0,0)$	M_3	p_3
$e(1,0,0)$	$(p_1-p_3-x-2\delta,0,0)$	M_4	δ
$l(1,0,0)$	(0,0,0)	M_1	$\max(3\delta,p_1-p_3-x-2\delta)=w$

Table 4.3: Case 2.a

set	remaining proc. time	r.p.	transition time
$l(1,0,0)$	(0,0,0)	M_1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$e(1,0,1)$	$(p_1 - x - \delta, 0, p_3)$	M_3	$5\delta + x$
$l(1,0,1)$	$(0, 0, p_3 - w')$	M_1	$\max(2\delta, p_1 - x - \delta) = w'$

Table 4.4: Case 2.b

Consider now the case where $l = 1$. This case is similar to the previous one until the cell reaches a cell state in $e(1,0,1)$ for the first time. Then, the scenario in Table 4.4 materializes up to state $l(1,0,1)$.

Thereafter, the robot performs the downhill schedule 011 111 110 101 and comes back to a cell state in $e(1,0,1)$ with remaining processing times $(\alpha, 0, p_3)$, where $\alpha \leq p_1 - 6\delta$. The transition time from $l(1,0,1)$ back to $e(1,0,1)$ amounts to at least 10δ. Thus, Table 4.4 can be extended as shown in Table 4.5 (in the last line of Table 4.5, we have used the fact that $\alpha \leq p_1 - 6\delta$ and $p_1 \leq 8\delta$).

set	remaining proc. time	r.p.	transition time
$l(1,0,0)$	(0,0,0)	M_1	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$e(1,0,1)$	$(p_1 - x - \delta, 0, p_3)$	M_3	$5\delta + x$
$l(1,0,1)$	$(0, 0, p_3 - w')$	M_1	$\max(2\delta, p_1 - x - \delta) = w'$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$e(1,0,1)$	$(\alpha, 0, p_3)$	M_3	10δ (at least)
$l(1,0,1)$	$(\alpha - p_3, 0, 0)$	M_3	p_3
$e(1,0,0)$	$(\alpha - p_3 - \delta, 0, 0)$	M_4	δ
$l(1,0,0)$	(0,0,0)	M_1	$\max(3\delta, \alpha - p_3 - \delta) = 3\delta$

Table 4.5: Case 2.b continued

So, for $l = 1$, it follows from Table 4.5 that the total cycle time is at least

$$\frac{19\delta + p_3 + x + w'}{2},$$

where as defined above $x = \max(\delta, p_2 - 3\delta)$ and $w' = \max(2\delta, p_1 - x - \delta)$.

Now, if $w' > 4\delta$, the proposed sequence is again dominated by the downhill schedule. Hence, we can assume that $w' \leq 4\delta$, from which we deduce $p_1 - x - \delta \leq 4\delta$ and, in turn, $w = \max(3\delta, p_1 - p_3 - x - 2\delta) = 3\delta$.

Thus, for $l = 0$, the cycle time is $9\delta + p_3 + x$. Moreover,

$$\begin{aligned} 19\delta + p_3 + x + w' &= 12\delta + 7\delta + p_3 + x + w' \\ &\geq 12\delta + 7\delta + p_3 + x + 2\delta \\ &= 12\delta + (9\delta + p_3 + x). \end{aligned}$$

This immediately implies that, if the sequence obtained for $l = 1$ (Case 2.b) is optimal, then so is the sequence obtained for $l = 0$ (and, simultaneously, so is the downhill sequence).

Finally, the above conclusion applies again when $l > 1$, since each additional execution of the downhill sequence requires time at least 12δ. ∎

Observation 4.2 In case 3, the sequence of loaded/unloaded states following $e(0,0,1)$ in the optimal sequence is one of the following :

001 (101 011 111/010 110)* 101 011 010,

Lemma 4.3 In case 3, the sequence of loaded/unloaded states following $l(1,0,0)$ in some optimal sequence is :

001 101 011 010.

Proof. First of all, it follows from Lemma 4.1, that the type of sequences as proposed in Observation 4.2 can be assumed not to contain more than one cell states in the set $e(0,1,0)$ (see the proof of Lemma 4.2). The remainder of this proof is also along the same lines as the proof of Lemma 4.2.

Informally, the proof establishes that the total travel time will turn out again to amount to $10\delta + l \times 12\delta$, where l is the number of times the downhill schedule is repeated. Thus, if we can show that by increasing l the total waiting time cannot decrease, we arrive in one of the following two cases. For $l = 0$ the waiting time is at least 2δ, but then the proposed sequence is dominated by the downhill sequence. On the other hand, if for $l = 0$ the waiting time is at most 2δ, the cycle time is minimized by setting $l = 0$, which again yields a 1-unit cycle. So let us examine the total waiting time. In case $l = 0$, the scenario of Table 4.6 materializes.

Thus, the total cycle time amounts to $7\delta + p_1 + w + v$ in case $l = 0$. Now, for $l = 1$, the waiting of p_1 cannot be avoided : until here both sequences are the same from the initial cell state in $e(0,0,1)$ as can be seen from Table 4.7. From $e(0,1,1)$, let the robot proceed to $e(1,0,1)$ by making the transitions specified by the downhill schedule. At this point, let α be the remaining processing time r_1 of machine M_1. It is the case that $\alpha \leq p_1 - 6\delta$ (because the robot travels at least 6δ since loading M_1). Clearly, $r_2 = 0$ and, since the robot just left $l(1,1,0)$, $r_3 = p_3$, and the robot is at M_3, see Table 4.8. It follows from Table 4.8 that for $l = 1$, the cycle time amounts to $19\delta + p_1 + w' + v'$. Now, if we can show that $w' + v' \geq w + v$, we have that

$$\begin{aligned} 19\delta + p_1 + w' + v' &\geq 19\delta + p_1 + w + v \\ &= 12\delta + 7\delta + p_1 + w + v, \end{aligned}$$

set	remaining proc. time	r.p.	transition time
$e(0,0,1)$	$(0,0,p_3)$	M_3	0
$l(0,0,1)$	$(0,0,p_3-3\delta)$	M_0	3δ
$e(1,0,1)$	$(p_1,0,p_3-4\delta)$	M_1	δ
$l(1,0,1)$	$(0,0,p_3-4\delta-p_1)$	M_1	p_1
$e(0,1,1)$	$(0,p_2,p_3-5\delta-p_1)$	M_2	δ
$l(0,1,1)$	$(0,p_2-w,0)$	M_3	$\max(\delta,p_3-p_1-5\delta)=w$
$e(0,1,0)$	$(0,p_2-w-\delta,0)$	M_4	δ
$l(0,1,0)$	$(0,0,0)$	M_2	$\max(2\delta,p_2-w-\delta)=v$
$e(0,0,1)$	$(0,0,p_3)$	M_3	δ

Table 4.6: Case 3.a

which suffices to prove the lemma (cf. the proof of Lemma 4.2). To see that $w'+v' \geq w+v$, notice that Tables 4.7 and 4.9 imply that $w' \geq w$ and $w'+v' = \max(2\delta+w', p_2-\delta)$ and $w+v = \max(2\delta+w, p_2-\delta)$.

Again, if $l > 1$, the same conclusion applies (cf. the proof of Lemma 4.2), since additional repetitions of the downhill permutations all require time at least 12δ. ∎

set	remaining proc. time	robot position	transition time
$e(0,0,1)$	$(0,0,p_3)$	M_3	0
$l(0,0,1)$	$(0,0,p_3-3\delta)$	M_0	3δ
$e(1,0,1)$	$(p_1,0,p_3-4\delta)$	M_1	δ
$l(1,0,1)$	$(0,0,p_3-4\delta-p_1)$	M_1	p_1
$e(0,1,1)$	$(0,p_2,p_3-5\delta-p_1)$	M_2	δ

Table 4.7: Case 3.b

Observation 4.3 In case 4, the sequence of loaded/unloaded states between $e(1,0,0)$ and $l(0,0,1)$ in the optimal sequence is one of the following :

100 010

100 010 110 (101 011 111/010 110)* 101 011 010.

It follows again from Lemma 4.1 that we are not interested in sequences in which cell states in the set $e(0,1,0)$ are entered more than twice.

set	remaining proc. time	robot position	transition time
$e(1,0,1)$	$(\alpha, 0, p_3)$	M_3	9δ (at least)
$l(1,0,1)$	$(0, 0, p_3 - 2\delta)$	M_1	$\max(2\delta, \alpha) = 2\delta$
$e(0,1,1)$	$(0, p_2, p_3 - 3\delta)$	M_2	δ
$l(0,1,1)$	$(0, p_2 - w', 0)$	M_3	$\max(\delta, p_3 - 3\delta) = w'$
$e(0,1,0)$	$(0, p_2 - w' - \delta, 0)$	M_4	δ
$l(0,1,0)$	$(0,0,0)$	M_2	$\max(2\delta, p_2 - w' - \delta) = v'$
$e(0,0,1)$	$(0,0,p_3)$	M_3	δ

Table 4.8: Case 3.b continued

Observation 4.4 In case 4, the sequence of loaded/unloaded states between $l(0,0,1)$ and $e(1,0,0)$ in the optimal sequence is one of the following :

001 000

001 (101 011 111/010 110)* 101

It follows again from Lemma 4.1 that we are not interested in sequences in which cell states in the set $e(0,1,0)$ are entered.

Thus we have in case 4 that some optimal sequence is one of the following :

100 010 001 000

100 010 001 (101 011 111 110)* 101

100 010 110 (101 011 111 110)* 101 011 010 001 000

100 010 110 (101 011 111 110)* 101 011 010 001 101 (011 111 110)* 101.

The first of these four sequences corresponds to the uphill permutation. We are going to show that, should case 4 apply, the uphill permutation must be optimal :

Lemma 4.4 In case 4, in some optimal sequence, the sequence of loaded/unloaded states is

100 010 001 000.

Proof. We first consider the third of the aforementioned sequences, sequence

100 010 110 (101 011 111 110)^l 101 011 010 001 000,

and show that it does not yield a uniquely optimal solution. This sequence is simulated in Table 4.9, for the case where $l = 0$.

set	remaining proc. time	r.p.	transition time
$e(1,0,0)$	$(p_1,0,0)$	M_1	0
$l(1,0,0)$	$(0,0,0)$	M_1	p_1
$e(0,1,0)$	$(0,p_2,0)$	M_2	δ
$l(0,1,0)$	$(0,p_2-2\delta,0)$	M_0	2δ
$e(1,1,0)$	$(p_1,p_2-3\delta,0)$	M_1	δ
$l(1,1,0)$	$(p_1-w,0,0)$	M_2	$\max(\delta,p_2-3\delta)=w$
$e(1,0,1)$	$(p_1-\delta-w,0,p_3)$	M_3	δ
$l(1,0,1)$	$(0,0,p_3-x)$	M_1	$\max(2\delta,p_1-\delta-w)=x$
$e(0,1,1)$	$(0,p_2,p_3-\delta-x)$	M_2	δ
$l(0,1,1)$	$(0,p_2-y,0)$	M_3	$\max(\delta,p_3-\delta-x)=y$
$e(0,1,0)$	$(0,p_2-\delta-y,0)$	M_4	δ
$l(0,1,0)$	$(0,0,0)$	M_2	$\max(2\delta,p_2-\delta-y)=z$
$e(0,0,1)$	$(0,0,p_3)$	M_3	δ
$l(0,0,1)$	$(0,0,0)$	M_3	p_3
$e(0,0,0)$	$(0,0,0)$	M_4	δ
$l(0,0,0)$	$(0,0,0)$	M_0	4δ
$e(1,0,0)$	$(p_1,0,0)$	M_1	δ

Table 4.9: Case 4

From Table 4.9, we see that the total execution time amounts at least $20\delta+p_1+p_3$. Hence if $p_1 > 4\delta$ or $p_3 > 4\delta$, then the downhill sequence is better. Otherwise, $x = 2\delta$ and $y = \delta$ and it follows that the total execution time amounts to

$$\begin{aligned} 14\delta + p_1 + p_3 + w + x + y + z &= 17\delta + p_1 + p_3 + w + z \\ &= 20\delta + p_1 + p_3 + 2\max(0, p_2 - 4\delta) \\ &= 10\delta + p_1 + \max(0, p_2 - 4\delta) + \\ &\quad 10\delta + p_3 + \max(0, p_2 - 4\delta). \end{aligned}$$

But this implies that, should this cycle be optimal and thus should case 4 apply, then both the 1-unit cycles of cases (and Lemmas) 4.2 and 4.3 are also optimal.

The reader can check that Table 4.10, which addresses the subcase $l = 1$, yields a cycle time of $\frac{26\delta+p_1+p_3+w+v+y+z}{3}$. Since $w+v+y+z \geq 6\delta$, it must be that $p_1 \leq 4\delta$ and $p_3 \leq 4\delta$, otherwise the downhill permutation has smaller cycle time. This implies $v = 2\delta$ and $y = \delta$. Thus, the cycle time amounts to $\frac{29\delta+p_1+p_3+w+z}{3} = \frac{32\delta+p_1+p_3+2\max(0,p_2-4\delta)}{3}$. Again we have that the optimal solutions of case 1 (Lemma 4.1), case 2 (Lemma 4.2) and case 3 (Lemma 4.3) have cycle time of resp. 12δ, $10\delta + p_3 + \max(0, p_2 - 4\delta)$, $10\delta + p_1 + \max(0, p_2 - 4\delta)$, if (in Table 4.6 $w = 0$, which follows here from) $p_3 \leq 4\delta$. It then follows that the minimum of these three cycle times does not exceed $\frac{32\delta+p_1+p_3+2\max(0,p_2-4\delta)}{3}$. In case $l > 1$, the analysis is similar, since additional executions

of a downhill schedule (which is the result of increasing l), require at least 12δ time units.

set	remaining proc. time	r.p.	transition time
$e(1,0,0)$	$(p_1,0,0)$	M_1	0
$l(1,0,0)$	$(0,0,0)$	M_1	p_1
$e(0,1,0)$	$(0,p_2,0)$	M_2	δ
$l(0,1,0)$	$(0,p_2-2\delta,0)$	M_0	2δ
$e(1,1,0)$	$(p_1,p_2-3\delta,0)$	M_1	δ
$l(1,1,0)$	$(p_1-w,0,0)$	M_2	$\max(\delta,p_2-3\delta)=w$
$e(1,0,1)$	$(p_1-\delta-w,0,p_3)$	M_3	δ
$l(1,0,1)$	$(0,0,p_3-x)$	M_1	$\max(2\delta,p_1-\delta-w)=x$
$e(0,1,1)$	$(0,p_2,p_3-x-\delta)$	M_2	δ
$l(0,1,1)$	$(0,p_2-2\delta,p_3-x-3\delta)$	M_0	2δ
$e(1,1,1)$	$(p_1,p_2-3\delta,p_3-x-4\delta)$	M_1	δ
$l(1,1,1)$	$(p_1-2\delta,p_2-5\delta,0)$	M_3	$\max(2\delta,p_3-x-4\delta)=2\delta$
$e(1,1,0)$	$(p_1-3\delta,p_2-6\delta,0)$	M_4	δ
$l(1,1,0)$	$(p_1-5\delta,0,0)$	M_2	$\max(2\delta,p_3-6\delta)=2\delta$
$e(1,0,1)$	$(p_1-6\delta,0,p_3)$	M_3	δ
$l(1,0,1)$	$(0,0,p_3-2\delta)$	M_1	$\max(2\delta,p_1-6\delta)=2\delta$
$e(0,1,1)$	$(0,p_2,p_3-3\delta)$	M_2	δ
$l(0,1,1)$	$(0,p_2-y,0)$	M_3	$\max(\delta,p_3-3\delta)=y$
$e(0,1,0)$	$(0,p_2-\delta-y,0)$	M_4	δ
$l(0,1,0)$	$(0,0,0)$	M_2	$\max(2\delta,p_2-\delta-y)=z$
$e(0,0,1)$	$(0,0,p_3)$	M_3	δ
$l(0,0,1)$	$(0,0,0)$	M_3	p_3
$e(0,0,0)$	$(0,0,0)$	M_4	δ
$l(0,0,0)$	$(0,0,0)$	M_0	4δ
$e(1,0,0)$	$(p_1,0,0)$	M_1	δ

Table 4.10: Case 4

The other two non 1-unit sequences of case 4 can be reasoned away along similar lines. Combining 100 010 and 001 $(101\ 011\ 111\ 110)^l$ 101 gives a sequence, in which all activities are executes $l+1$ times. If $l=0$, the cycle time is at least 12δ as can be checked. If $l=1$, the total cycle time in this sequence is at least $12\delta+8\delta+p_1+p_2+p_3$, which yields it to be dominated, since 12δ (Lemma 4.1) and $8\delta+p_1+p_2+p_3$ (Lemma 4.4) are attainable 1-unit cycle cycle times and :

$$\frac{12\delta+8\delta+p_1+p_2+p_3}{2} \geq \min(12\delta, 8\delta+p_1+p_2+p_3).$$

Again increasing l by one (from at least 1) adds 12δ per additional execution of the downhill permutation.

A similar argument holds in the remaining subcase, in which the two sequences 100 010 110 (101 011 111 110)^k 101 011 010 and 001 (101 011 111 110)^l 101 are combined. If $k = l = 0$, the total travel time is already 24δ. If $k = 0$ and $l = 1$, the cycle time amounts to at least $\frac{32\delta+p_1+p_3+2\max(0,p_2-4\delta)}{3}$, as in subcase 1 of case 4. Additional executions of the downhill sequence (which arise from increasing k or l by 1), require at least 12δ time units. ■

We finally arrive at the main result of this chapter :

Theorem 4.1 In a 3 machine robotic flowshop, 1-unit cycles are optimal in case of identical parts when, $\delta_i = \delta$ for $i = 0, 1, 2, 3$, and $\epsilon_i = 0$ for $i = 0, 1, 2, 3, 4$.

Proof. The theorem is a consequence of Lemmas 4.1-4.4, that cover all possible cases.■

4.4 Generalisations and further research

There are several directions possible for generalising the results of the previous section. A first generalisation would be to consider the case that arises when dropping the assumption of zero loading/unloading times and equidistant machines, i.e. $\delta_i = \delta$ for $i = 1, \ldots, 4$, $\epsilon_i = 0$ for $i = 0, \ldots, 4$. A straightforward but rather tedious extension of the arguments in the previous section shows that all results go through for the generalisation with machine dependent loading/unloading times and non-equidistant machines.

A second generalisation is to prove the conjecture of Sethi et al. [1992]. We show that 1-unit cycles have minimum cycle time over all finite length robot move sequences, leaving open whether there can exist infinite sequences that attain a lower cycle time. We conjecture with Sethi et al. [1992] that no infinite sequence can attain a smaller cycle time than the smallest cycle time attainable by a 1-unit cycle.

A third, and most challenging generalisation, would be to extend the results to robotic cells with more than three machines. Indeed, we propose an extension of the conjecture of Sethi et al. [1992] to the case of cells with an arbitrary number of machines. As a first step, one could further investigate the relationship with pyramidal permutations that we have established in the previous section.

Part II

Printed circuit board assembly

Chapter 5

The component retrieval problem in printed circuit board assembly

5.1 Introduction

The problem of determining optimal production plans for the automated assembly of printed circuit boards (PCBs) has been investigated by numerous researchers; see e.g. Ahmadi [1993] and Crama, Oerlemans and Spieksma [1994]. A precise definition of this problem is highly dependent on the specific features of the assembly machines and, more generally, of the technological environment. As a rule, however, the problem is a very complex one. For this reason, many authors have proposed to solve it by decomposing it into subproblems; see e.g. Ahmadi [1993], Ball and Magazine [1988], Bard, Clayton and Feo [1994], Crama, Flippo, van de Klundert and Spieksma [1995b], Crama, Oerlemans and Spieksma [1994] and van Laarhoven and Zijm [1993]. Here again, the subproblems emerging from the decomposition vary according to the context, as do their computational complexity. In this chapter, we concentrate on one such subproblem, namely the Component Retrieval Problem (CRP) for PCB assembly with a placement machine that operates like the Fuji CP II. Briefly stated, CRP is defined to be the following problem: for a given placement sequence of components on the board, and for a given assignment of component types to (possibly multiple) feeder slots of the placement machine, decide from which feeder slots the components should be retrieved.

In the next section, we describe the assembly process with a Fuji CP II placement machine, and the role of the Component Retrieval Problem in this process. We refer to Bard, Clayton and Feo [1994] and Crama et al. [1995b] for a more complete description of this process as well as for related references. In Section 5.3, we present a formulation of CRP in terms of a PERT/CPM network problem with design aspects; finding the minimal makespan of the assembly process thus amounts to identifying a design for which the longest path in the induced network is shortest. Alternatively, the Component Retrieval Problem may also be viewed as a shortest path problem with (path induced) side-constraints. In Section 5.4 an example is discussed which reveals

that straightforward forward dynamic programming does not necessarily yield optimal solutions. In fact, the example suggests that no dynamic programming approach with a linearly sized state-space in the number of components involved, is capable of retaining sufficient information to identify optimal solutions in all cases. These negative observations, which also invalidate the forward dynamic programming approach by Bard, Clayton and Feo [1994], may serve as a justification for the relatively complex solution algorithm that is proposed in Section 5.5. The algorithm is based on the network formulation of Section 5.3, and can be viewed as a "two-phase" dynamic programming approach with *pairs* of grip activities as the state-space. If the number of components involved is denoted by n, then the size of the state-space is thus *quadratic* in n, implying an overall time complexity of $\mathcal{O}(n^3)$ for the entire algorithm.

Since the network optimization problems of Section 5.3 are of interest beyond the special case of CRP, their time complexity is studied in Section 5.6. It is proven that the polynomial solvability of the Component Retrieval Problem is caused by the specific structure it inflicts on the arc lengths in the PERT/CPM network; in the absence of this structure, the network problems are shown to be *NP*-hard in general. The chapter is concluded with a brief summary.

5.2 The Fuji CP II placement machine

Assembling a printed circuit board consists of placing a number of electronic components, each of prespecified type, at prespecified locations on a bare board. The placement machine that is considered in this chapter is a Fuji CP II, yet our analysis may apply to other machines having similar characteristics as well (such as other members of the Fuji family, or the Panasonic Mk1 considered by Horak and Francis [1995]). The Fuji CP II is equipped with a magazine rack that contains a number of slots to which feeder tapes can be assigned. Each tape bears components of a unique component type, and feeder tapes with the same component type may be assigned to multiple slots. (In fact, it will become clear shortly that a non-trivial instance of the Component Retrieval Problem only emerges if at least one component type is assigned to at least two different feeder slots.) Components are gripped from a slot of the magazine rack and mounted on the PCB by a placement head. Coordination between grip and place activities is done by a carousel, which performs many other functions as well. The carousel contains 12 heads, and it can simultaneously hold up to six components; see Figure 5.1.

Suppose the machine is just about to place the i-th component on the board. To this end, the board location where the component is to be placed, is positioned at the so-called "placement spot", and the carousel head containing the component (the current place station), is right above this spot. The carousel head then proceeds with the actual placement of the component on the board. After the placement has been completed, the worktable holding the PCB starts moving in order to position the board location where the next component is to be placed, at the placement spot.

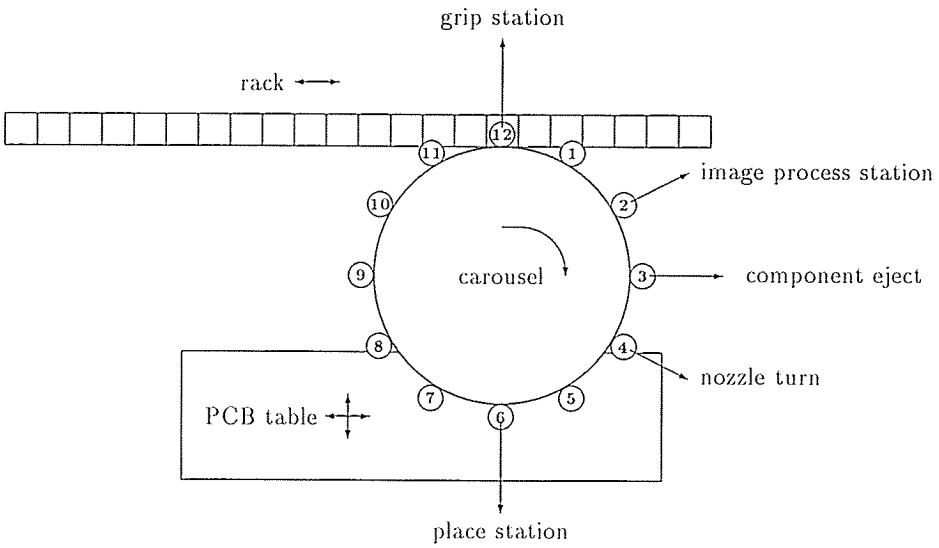

Figure 5.1: The Fuji CP II.

Diametrically opposed to the aforementioned carousel head is another head (the current grip station), which is positioned right above the so-called "gripping spot". The grip station is ready to grip the $(i + 6)$-th component from the magazine rack as soon as the appropriate slot has been positioned at the gripping spot. Once this is done, the head proceeds with actually gripping the $(i + 6)$-th component from this feeder slot. After the gripping activity has been completed, the magazine rack will start to shift in order to position the slot from which the next component is to be retrieved, at the gripping spot. Only after the i-th component has been placed and the $(i + 6)$-th component has been gripped, the carousel is ready to rotate 30° clockwise, to prepare for the placement and gripping of component $(i + 1)$ and $(i + 7)$ respectively.

Thus, between two consecutive place activities, the PCB table has to move until the location where the second component is to be placed at the board, is at the placement spot, and the carousel has to rotate 30° so as to position the next head right above the placement spot. Similarly, between two consecutive grip activities, the rack has to shift until the appropriate slot is at the gripping spot, and the carousel has to rotate so as to position the next head right above this spot. It is hereby important to observe that placement operation i and gripping operation $(i + 6)$ do not have to be performed simultaneously, but are necessarily carried out between the same two carousel rotations. Also, table and rack movements may take place concurrently with each other, and with a carousel rotation.

Clearly, the duration of rack movements depends on the distance between slots from which consecutive gripping operations are done. Therefore, even when the component placement sequence on the board and the magazine rack assignment of component

tapes are given, minimizing the assembly makespan still involves decisions concerning the feeder slots from which the components should be retrieved. These decisions are collectively known as the *Component Retrieval Problem (CRP)*. As has been stated before, a non-trivial decision problem only emerges if at least one component type is assigned to at least two different feeder slots. This type of feeder duplication is also discussed in e.g. Ahmadi, Grotzinger and Johnson [1988], Bard, Clayton and Feo [1994] and Tang and Denardo [1988].

5.3 The Component retrieval problem as a PERT/CPM network model with design aspects

In order to facilitate our discussion, we present a PERT/CPM-like model of CRP. To achieve this, we first need to introduce several assumptions and develop some notation. First, let $1, \ldots, n$ denote the components that are to be mounted on the PCB, with the numbering reflecting their placement sequence on the board. With respect to the starting conditions of the assembly process, we assume that the feeder slot from which the first component will be retrieved, is initially positioned below the grip station (currently occupied by carousel head 12), and that the PCB location where the first place activity will occur, is initially positioned below the place station (currently occupied by carousel head 6). Furthermore, components 1–6 have been added as fictitious components, initially held by carousel heads 6–1 respectively; they are to be mounted at the same board location as component 7, which operation can be performed in zero time. If we similarly assume that 6 fictitious and instantaneous grip activities are carried out at the end of the mounting process, then a situation has been constructed where exactly n grip activities and n place activities are required to assemble the board, with the i-th grip and i-th place activity occurring between the $(i-1)$-st and i-th carousel rotation.

As a first step towards modeling CRP, let us briefly recall how, for a given solution S of CRP, the assembly makespan can be computed by classical PERT/CPM techniques. To this end, the *events* (i.e. moments in time) and *activities* (i.e. time durations) of Table 5.1 are introduced.

Events, activities and precedence relations between activities can be represented by a PERT/CPM graph $D(S)$ (recall that S is the given solution to CRP), where nodes and arcs correspond to events and activities respectively, and arc lengths denote activity durations; see Figure 5.2. We will refer to the nodes as grip or place nodes, depending on the nature of the associated event. The resulting graph consists of n layers, where each layer i contains exactly one grip node g_i and one place node p_i $(i = 1, \ldots, n)$. To model the start and the end of the assembly process, it is convenient to add a source, indifferently denoted by p_0 and g_0, and a sink, indifferently denoted by p_{n+1} and g_{n+1}. As is well-known, the makespan of the assembly process is equal to

g_i	=	start of the i-th grip activity	$(i = 1, \ldots, n)$
p_i	=	start of the i-th place activity	$(i = 1, \ldots, n)$
Δg_i	=	duration of the i-th grip activity	$(i = 1, \ldots, n)$
Δp_i	=	duration of the i-th place activity	$(i = 1, \ldots, n)$
Δm_i	=	duration of the i-th rack movement	$(i = 1, \ldots, n-1)$
Δt_i	=	duration of the i-th table movement	$(i = 1, \ldots, n-1)$
Δc_i	=	duration of the i-th carousel rotation	$(i = 1, \ldots, n-1)$

Table 5.1: Events and activities of the PERT/CPM graph $D(S)$

the length of a longest path in $D(S)$ from p_0 to p_{n+1}. Computing a longest path in such an acyclic and layered network can be done by forward dynamic programming in $\mathcal{O}(n)$ time (see e.g. Ahuja, Magnanti and Orlin [1993]).

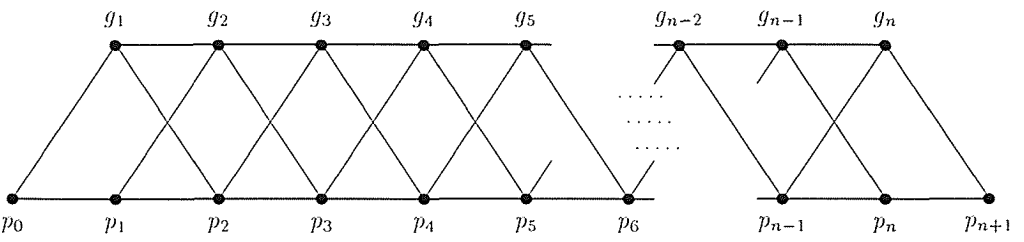

Figure 5.2: The PERT/CPM graph $D(S)$

In order to specify the arc lengths of $D(S)$, recall from Section 5.2 that between the start of two consecutive grip activities g_i and g_{i+1} $(i = 1, \ldots, n-1)$, the following operations have to be performed: the i-th grip activity, the i-th carousel rotation and the i-th rack movement. Since the former precedes the latter two, and the latter two may be carried out concurrently, it follows that the length of arc (g_i, g_{i+1}) equals $\Delta g_i + \max\{\Delta c_i, \Delta m_i\}$. On the other hand, the $(i+1)$-st grip activity can only start when both the i-th place activity and the i-th carousel rotation are completed. Since these activities are carried out consecutively, it follows that the length of arc (p_i, g_{i+1}) equals $\Delta p_i + \Delta c_i$. Other arc lengths in $D(S)$ are defined in a similar fashion; see Table 5.2.

In view of the above discussion, the Component Retrieval Problem can now be modelled as follows. Consider the graph of Figure 5.2. For each $i = 1, \ldots, n$, we introduce a *set* of grip nodes G_i instead of only one grip node g_i, where each node of G_i refers to one of the slots containing the component type required for the i-th grip activity. Figure 5.3 shows an example where the first two components (which may or

arc	for	length
(g_i,g_{i+1})	$i = 1,\ldots,n-1$	$\Delta g_i + \max\{\Delta c_i, \Delta m_i\}$
(p_i,p_{i+1})	$i = 0,\ldots,n$	$\Delta p_i + \max\{\Delta c_i, \Delta t_i\}$
(g_i,p_{i+1})	$i = 1,\ldots,n$	$\Delta g_i + \Delta c_i$
(p_i,g_{i+1})	$i = 0,\ldots,n-1$	$\Delta p_i + \Delta c_i$

Table 5.2: Arc lengths of $D(S)$ with $\Delta c_i = \Delta t_i = 0$ for $i = 0, n$ and $\Delta p_0 = 0$

may not be of the same type) can both be retrieved from two alternative feeder slots, and all other components can only be retrieved from one such slot. Then specifying a component retrieval plan, i.e. a feasible solution S of CRP, amounts to selecting exactly one grip node from each set G_i, such that the longest path in the induced subgraph is shortest. We thus arrive at the following formalization of CRP graphs and problems.

Definition 5.1 A *CRP graph* $D = (V, A)$ is a layered directed graph on the node set $V = \cup_{i=0}^{n+1} L_i$, with the layers L_i being mutually disjoint sets. Moreover, $L_i = \{p_i\} \cup G_i$, where G_i is a non-empty set not containing p_i $(i = 1,\ldots,n)$ and $G_0 = G_{n+1} = \emptyset$. The set $\{p_0,\ldots,p_{n+1}\}$ is referred to as the set of place nodes; all other nodes are called grip nodes. The arc set A is given by $A = \{(u,v) | u \in L_i,\ v \in L_{i+1}$ for some $i = 0,\ldots,n\}$, with the length of arc (u,v) being denoted by $d(u,v)$. The length function $d(\cdot)$ satisfies

$$d(g_i,p_{i+1}) + d(p_i,g_{i+1}) \leq d(g_i,g_{i+1}) + d(p_i,p_{i+1}). \tag{5.1}$$

for all $g_i \in G_i$, $g_{i+1} \in G_{i+1}$ and $i = 1,\ldots,n-1$.

Note that the arc lengths displayed in Table 5.2 satisfy (5.1). In the sequel, however, we will *not* make any explicit use of the specific lengths in Table 5.2, but rely on their property (5.1) instead. We will see in Sections 5.5 and 5.6 that this property guarantees the efficient solvability of CRP. It may also be interesting to remark that the inequalities (5.1) are somewhat reminiscent of a matrix property studied in the literature under the name 'Monge property' (see e.g. Burkard, Klinz and Rudolf [1995]).

Definition 5.2 A *selection* S in a CRP graph D is a set of grip nodes containing exactly one grip node from each layer, i.e. $|S \cap G_i| = 1$ for $i = 1,\ldots,n$.

Definition 5.3 For any selection S in a CRP graph D, the *selection induced subgraph* $D(S) = (V(S), A(S))$ is the subgraph of D that is induced by $S \cup \{p_0,\ldots,p_{n+1}\}$. The length of a longest path in $D(S)$ is denoted by $L(D(S))$.

Since the length of a longest path in a selection induced subgraph is equal to the makespan of the PCB assembly process using the specified selection of feeder slots, we arrive at the following network version of the Component Retrieval Problem.

Definition 5.4 Given a CRP graph D, the *Component Retrieval Problem* is to determine a selection S such that the longest path length of $D(S)$ is shortest.

As mentioned before, the analysis and results in this chapter will all apply to this network version of the Component Retrieval Problem, and not only to the special instances arising from the original application where arc lengths are as in Table 5.2.

As a final remark, the aforementioned definitions reveal that CRP is basically a PERT/CPM network problem with design aspects. Obviously, designs are restricted to selections in this case, i.e. they must contain exactly one grip node per layer. As an alternative interpretation, the minimization of the makespan seems to indicate that all grip activities should be completed as early as possible. This, in its turn, seems to suggest that a minimal makespan can be obtained by computing a shortest path from p_0 to p_{n+1} in the subgraph that is induced by these two place nodes and all grip nodes (the so-called "grip graph"). Unfortunately, the precedence relations that are induced by the place activities would be completely ignored in such an approach; a shortest path through the grip graph would only specify an optimal selection if, between each pair of grip nodes, the makespan (longest path length) that results from the interfering place activities (nodes) was taken into consideration as a lower bounding side-constraint. In fact, Section 5.5 reveals that this is exactly the approach of our solution algorithm. Therefore, CRP can also be viewed as a shortest path problem with side-constraints. Obviously, the side-constraints are of a very specific nature here, viz. they result from longest path lengths induced by a single (place) path that is added to the (grip) graph under consideration. The aforementioned two interpretations of CRP are interesting in their own right. In Section 5.6 it will be shown that although CRP can be solved in polynomial time, more general versions of the problem probably cannot, since the absence of the arc length structure (5.1) makes them *NP*-hard in general.

5.4 An example why straightforward dynamic programming does not work

Before we present our algorithm for CRP, it may be instructive to first discuss an example that indicates why CRP is not a trivial problem, in the sense that it can*not* be solved by a straighforward dynamic programming scheme. More specifically, let D be a CRP graph and S be an optimal selection. For some k, let $V_k = (\cup_{i=0}^{k} L_i) \cup \{g_{k+1}\}$. Let D_k be the CRP subgraph of D induced by V_k, and let $S_k = S \cap V_k$. Then it is generally *not* true that S_k is an optimal selection for D_k. The same remark applies to $(\cup_{i=0}^{k} L_i) \cup \{p_{k+1}\}$.

Consider the CRP graph of Figure 5.3. Let the arc lengths of (g_1^1, g_2^2) and (g_1^2, g_2^1) be "large" (say ≥ 10), and all (p_{i-1}, g_i) and (g_i, p_{i+1}) $(i = 1, \ldots, n)$ have length zero. The other arc lengths are as indicated. Note that (g_{n-1}, g_n) has length $4 + x$. We will consider two possible values for x, viz. $x = 0$ and $x = 5$ respectively.

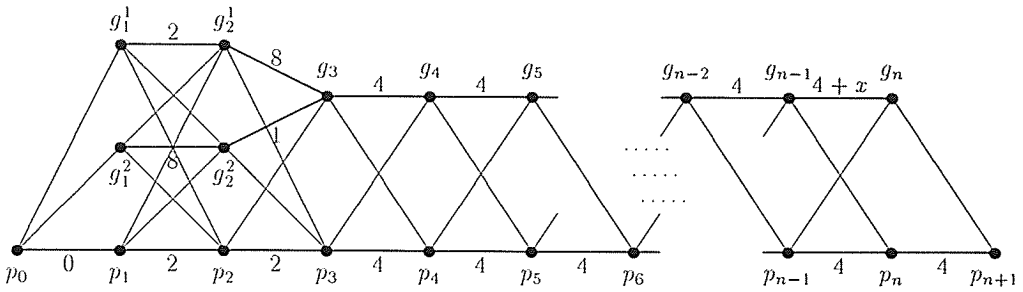

Figure 5.3: A counterexample for straightforward dynamic programming

Longest Paths	S^1	S^2
$x = 0$	$\{p_0, g_1^1, g_2^1, g_3, \ldots, g_n, p_{n+1}\}$ length: $10 + 4(n-3)$	$\{p_0, g_1^2, g_2^2, p_3, \ldots, p_n, p_{n+1}\}$ length: $12 + 4(n-3)$
$x = 5$	$\{p_0, g_1^1, g_2^1, g_3, \ldots, g_n, p_{n+1}\}$ length: $15 + 4(n-3)$	$\{p_0, g_1^2, g_2^2, g_3, \ldots, g_n, p_{n+1}\}$ length: $14 + 4(n-3)$

Table 5.3: Longest path (length) under different scenarios

Since $d(g_1^1, g_2^2)$ and $d(g_1^2, g_2^1)$ are "large", the only candidate optimal selections are $S^1 = \{g_1^1, g_2^1, g_3, \ldots, g_n\}$ and $S^2 = \{g_1^2, g_2^2, g_3, \ldots, g_n\}$. Table 5.3 summarizes the longest path (length) in the corresponding selection induced subgraphs, for $x = 0$ and $x = 5$ respectively. Observe that S^1 is optimal when $x = 0$, whereas S^2 is optimal when $x = 5$.

A simple forward dynamic programming scheme with the grip nodes as its state space would be based on the following definition:

$$\phi(g_i) \quad = \quad \text{the shortest longest path length until } g_i$$

Note that $\phi(g_i) = 9 + 4(i-3)$ for $3 \leq i \leq n-1$, with the optimal predecessor of g_i being g_2^2. However, when $x = 0$ then g_2^2 is not part of any selection that is optimal for the entire assembly process. Similarly, if the place nodes form the state space, hence

$$\psi(p_i) = \text{the shortest longest path length until } p_i$$

then $\psi(p_i) = 6 + 4(i-3)$ for $4 \leq i \leq n$ with the optimal predecessor of p_i being g_2^1. Yet again, when $x = 5$ then g_2^1 is not part of any selection that is optimal for the entire assembly process. These observations clearly show that the Principle of Optimality does not hold in either case; in order to identify a partial selection that is part of an optimal selection for the entire problem, it may be necessary to keep track of partial

selections that are *non*-optimal up to certain layers. In addition, the information that is required to track the first part of an optimal selection for the entire problem, may be contained in arbitrarily remote parts of the graph, even as remote as the very last grip arc. The conclusion is that simple forward dynamic programming does not necessarily identify optimal selections, not even if the recursion is equipped with a "look-k-layers-ahead-or-back" capability for constant k. To guarantee optimality, a more elaborate analysis and approach therefore seems to be required.

As a final remark it is mentioned that this negative conclusion also affects the validity of the forward dynamic programming algorithm by Bard, Clayton and Feo [1994], in the sense that their method does *not* constitute an exact solution method for all problem instances. The recursive formula that is considered by these authors reads

$$f_i(k) = \min_{j \in Y_{i-1}} \{t_{jk}(i-1, i) + f_{i-1}(j)\} \qquad k \in Y_i,\ i = 1, \ldots, n$$

where

$f_i(k)$ = minimum time required to grip the first i components given that the i-th component is retrieved from magazine slot k;

Y_i = set of magazine slots containing the component type required for the i-th gripping activity;

$t_{jk}(i-1, i)$ = elapsed time between completion of the $(i-1)$-st gripping activity from slot j and the i-th gripping activity from slot k.

Note that the interpretation of $f_i(k)$ coincides with the previously mentioned $\phi(g_i)$. As the example of this section shows (and what seems to have been overlooked by these authors) is that in an *optimal* retrieval plan where the i-th component is retrieved from magazine slot k, the time required to grip the first i components may *strictly exceed* $f_i(k)$ for some i and k.

5.5 A polynomial algorithm for CRP

In this section, we consider a given CRP graph D, and we present a polynomial algorithm for CRP as formulated in Definition 5.4. As is explained in Section 5.3, the optimal selection in D can generally *not* be computed by solving for a shortest path in the subgraph that is induced by the grip nodes of D, since the side-constraints that are induced by the precedence relations of the interfering place activities, would be completely ignored in that case. The general approach in this section is to model each of these side-constraints as an arc between two grip nodes, with its length equal to the smallest longest path in D between these grip nodes. The optimal selection can then be retrieved by solving for the shortest path in this newly constructed graph, which will be denoted by $D_{\mathcal{N}}$. Since the arc lengths in $D_{\mathcal{N}}$ can be computed by a (polynomial and simple) forward dynamic programming approach, and since a shortest path in $D_{\mathcal{N}}$

can be computed likewise, our procedure can be thought of as a "two-phase" forward dynamic programming algorithm.

This section is built up as follows. First, a simplified version of CRP will be considered, which can be solved by forward dynamic programming in polynomial time. The insights that have thus been obtained will then be used to arrive at a polynomial algorithm for CRP itself. Application of the proposed algorithm to the numerical example of Section 5.4 will conclude this section.

5.5.1 A simplified version of CRP

Lemma 5.1 The length of the path $(p_0, p_1, \ldots, p_{n+1})$ through the place nodes is a lowerbound on the optimal solution value of CRP.

Proof. Straightforward. ∎

This simple observation motivates our interest in the following problem.

Definition 5.5 CRP*

INPUT: A CRP graph D;

QUESTION: Is there a selection S such that the path $(p_0, p_1, \ldots, p_{n+1})$ through the place nodes is a longest path of $D(S)$.

Note that, if the answer to CRP* is affirmative for some selection S, then S is an optimal solution to CRP (cf. Lemma 5.1). For $0 \leq i \leq j \leq n+1$, let $L_P(i,j)$ be the length of the path $(p_i, p_{i+1}, \ldots, p_j)$ from p_i to p_j through the place nodes. Similarly, for a selection $S = \{g_1, g_2, \ldots, g_n\}$ and for $j - i \geq 2$, let $L_G(S,i,j)$ be the length of the path $(p_i, g_{i+1}, \ldots, g_{j-1}, p_j)$ from p_i to p_j with all intermediates nodes in S.

Theorem 5.1 For every selection S, the path $(p_0, p_1, \ldots, p_{n+1})$ is a longest path of $D(S)$ if and only if $L_G(S,i,j) \leq L_P(i,j)$ for all $i,j \in \{0, \ldots, n+1\}$ with $j - i \geq 2$.

Proof. If $L_G(S,i,j) > L_P(i,j)$ for some i,j, then the path

$$(p_0, p_1, \ldots, p_i, g_{i+1}, \ldots, g_{j-1}, p_j, \ldots, p_{n+1})$$

is longer than the path through the place nodes. On the other hand, the inequalities in the theorem imply that the path through the place nodes will be at least as long as any path containing some grip nodes. ∎

Theorem 5.1 motivates the introduction of a collection of *s-labels* associated with each selection S, which reflect the slack that S displays with respect to the necessary and sufficient conditions stated in the theorem.

Definition 5.6 For every selection S and every $j \in \{1, 2, \ldots, n\}$, define

$$s(S, j) = \min_{0 \le i \le j-1} \{L_P(i, j+1) - L_G(S, i, j+1)\}. \tag{5.2}$$

We view label $s(S, j)$ as being attached to the j-th grip node of S. Theorem 5.1 can now be equivalently stated as follows.

Corollary 5.1 For every selection S, the path $(p_0, \ldots, p_{n+1})$ is a longest path of $D(S)$ if and only if

$$s(S, j) \ge 0 \quad \text{for all } j \in \{1, 2, \ldots, n\}. \tag{5.3}$$

The s-labels satisfy the following recursion:

Lemma 5.2 For every selection $S = \{g_1, g_2, \ldots, g_n\}$ and every $j \in \{2, 3, \ldots, n\}$,

$$\begin{aligned} s(S, j) = \min\{ & s(S, j-1) + L_P(j, j+1) + d(g_{j-1}, p_j) - d(g_{j-1}, g_j) - d(g_j, p_{j+1}), \\ & L_P(j-1, j+1) - d(p_{j-1}, g_j) - d(g_j, p_{j+1})\}. \end{aligned}$$

Proof. For each $i \in \{0, \ldots, j-2\}$, we can rewrite

$$\begin{aligned} & L_P(i, j+1) - L_G(S, i, j+1) = \\ & [L_P(i, j) + L_P(j, j+1)] - [L_G(S, i, j) - d(g_{j-1}, p_j) + d(g_{j-1}, g_j) + d(g_j, p_{j+1})]. \end{aligned}$$

The validity of the lemma follows directly from this observation and from Definition 5.6. ∎

Lemma 5.2 provides a recursive formulation of the s-labels associated for a *given* selection S. In order to solve CRP*, we now generalize the s-labels by introducing a label $s^*(g_j)$ attached to each grip node $g_j \in G_j$. The value of $s^*(g_j)$ is the largest value of $s(S, j)$ that can be attained by any selection S containing g_j and satisfying condition (5.3) up to layer $j-1$. More precisely,

Definition 5.7 For all $j \in \{1, 2, \ldots, n\}$ and all $g_j \in G_j$, let $T(g_j)$ denote the set of selections S with

(i). $g_j \in S$, and

(ii). $s(S, i) \ge 0$ for all $i \in \{1, \ldots, j-1\}$.

Then we define $s^*(g_j) = \max_{S \in T(g_j)} s(S, j)$.

As usual, we let $s^*(g_j) = -\infty$ when $T(g_j) = \emptyset$. Let us stress the following properties of the s^*-labels, which are direct consequences of Definition 5.7.

P1. $-\infty < s^*(g_j) < 0$ if and only if $T(g_j) \neq \emptyset$ and $s(S, j) < 0$ for every selection $S \in T(g_j)$.

P2. $s^*(g_j) \geq 0$ if and only if there exists a selection S with $g_j \in S$ and $s(S,i) \geq 0$ for all $i \in \{1,\ldots,j\}$.

In particular, combining these properties with Corollary 5.1 renders

Theorem 5.2 The answer to CRP* is affirmative if and only if $s^*(g_n) \geq 0$ for some node $g_n \in G_n$.

Similar to the s-labels, the s^*-labels can also be computed by dynamic programming (cf. Lemma 5.2).

Theorem 5.3 For all $j \in \{2,3,\ldots,n\}$ and for all $g_j \in G_j$,

$$s^*(g_j) = \max_{g_{j-1} \in G_{j-1}: s^*(g_{j-1}) \geq 0} \min\{\; s^*(g_{j-1}) + L_P(j,j+1) + d(g_{j-1},p_j) - d(g_{j-1},g_j) - d(g_j,p_{j+1}), \quad L_P(j-1,j+1) - d(p_{j-1},g_j) - d(g_j,p_{j+1})\}. \tag{5.4}$$

Proof. Fix $j \in \{2,3,\ldots,n\}$ and $g_j \in G_j$. Denote the right-hand side of (5.5) by σ.

(i). Assume first that $T(g_j) \neq \emptyset$. Then, by Definition 5.7, there exists an $S \in T(g_j)$ with $s^*(g_j) = s(S,j)$. If we write $S = \{g_1, g_2, \ldots, g_n\}$, then it is clear that $S \in T(g_{j-1})$, so $s(S,j-1) \leq s^*(g_{j-1})$. In addition, $s(S,j-1) \geq 0$. Combining the latter two inequalities with Lemma 5.2, renders $-\infty < s^*(g_j) = s(S,j) \leq \sigma$.

(ii). Conversely, assume now that $\sigma > -\infty$, and let $g_{j-1} \in G_{j-1}$ attain the maximum in the definition of σ, i.e. $\sigma = \min\{a,b\}$ with $a = s^*(g_{j-1}) + L_P(j,j+1) + d(g_{j-1},p_j) - d(g_{j-1},g_j) - d(g_j,p_{j+1})$ and $b = L_P(j-1,j+1) - d(p_{j-1},g_j) - d(g_j,p_{j+1})$. By Definition 5.7, there exists a selection $S \in T(g_{j-1})$ with $s^*(g_{j-1}) = s(S,j-1)$. Without loss of generality, we can assume that $g_j \in S$ (otherwise, substitute g_j for the j-th grip node of S). Then, by Lemma 5.2, $s(S,j) = \sigma$. On the other hand, since $s^*(g_{j-1}) \geq 0$, we deduce that $S \in T(g_j)$ and, by Definition 5.7,

$$\sigma = s(S,j) \leq \max_{R \in T(g_j)} s(R,j) = s^*(g_j).$$

Taken together, (i) and (ii) establish the theorem. ∎

Theorem 5.3 implies that the s^*-labels can be computed in polynomial time, layer by layer. In view of Theorem 5.2, we have thus obtained a polynomial algorithm for the solution of CRP*. This algorithm can be implemented to run in $O(e)$ time, where e is the number of arcs of D. Moreover, the proof of Theorem 5.3 establishes that, in addition to answering CRP*, we can also find a selection S with $s(S,j) \geq 0$ for all $0 \leq j \leq n$ if one exists. As a final remark, we observe that, up to this point, we have not made any use of the properties of arc lengths recorded in Definition 5.1. In other words, Theorem 5.3 applies for arbitrary arc lengths.

5.5.2 Further properties of the s-labels

We have just described the role that the s-labels play in solving CRP*. In the next subsection, these ideas will be incorporated into an algorithm for the full-fledged Component Retrieval Problem. In order to achieve this goal, we first need to understand some of the basic properties of the s-labels. These properties will now be recorded in a sequence of lemmas.

Lemma 5.3 For any $j \in \{1, 2, \ldots, n\}$ and $g_j \in G_j$, let $S = \{g_1, g_2, \ldots, g_n\}$ be a selection in $T(g_j)$. Consider $D(S)$. Then

(i). the path $(p_0, p_1, \ldots, p_j)$ is a longest path from p_0 to p_j with length $L_P(0, j)$;

(ii). the path $(p_0, p_1, \ldots, p_i, g_{i+1}, g_{i+2}, \ldots, g_j)$ is a longest path from p_0 to g_j with length

$$L_P(0, j+1) - s(S, j) - d(g_j, p_{j+1}). \tag{5.5}$$

where i is any index that realizes the minimum in the expression (5.2) defining $s(S, j)$.

Proof.

(i). By Definition 5.7, $s(S, i) \geq 0$ for all $i \in \{1, \ldots, j-1\}$. The claim is now a straighforward extension of Corollary 5.1.

(ii). Let P_{g_j} be any longest path from p_0 to g_j and let $k = \max\{\ell \mid p_\ell \in P_{g_j}\}$. Since $0 \leq k \leq j-1$, we have $S \in T(g_k)$, and hence $(p_0, p_1, \ldots, p_k)$ is a longest path from p_0 to p_k (cf. (i)). Thus, without loss of generality, we can assume that $P_{g_j} = (p_0, p_1, \ldots, p_k, g_{k+1}, \ldots, g_j)$. The length of P_{g_j} is now easily checked to be given by

$$L_P(0, j+1) - (L_P(k, j+1) - L_G(S, k, j+1)) - d(g_j, p_{j+1}).$$

In view of (5.2), the latter expression is maximized when $L_P(k, j+1) - L_G(S, k, j+1) = s(S, j)$, i.e. when $k = i$. Thus we may indeed conclude that the path $(p_0, p_1, \ldots, p_i, g_{i+1}, g_{i+2}, \ldots, g_j)$ is a longest path from p_0 to g_j with length as stated in (5.5). ∎

Next, let us consider what happens when, in a selection induced subgraph $D(S)$, the longest path is *not* the path of place nodes $(p_0, p_1, \ldots, p_{n+1})$ (this is the only interesting case, since we know from the previous subsection how to handle yes-instances of CRP*). In such a case, we already know by Corollary 5.1 that $s(S, j)$ must be negative for some layer j. Let us consider the first such layer.

Lemma 5.4 For any selection $S = \{g_1, g_2, \ldots, g_n\}$, let j be the smallest index in $\{1, 2, \ldots, n\}$ with $s(S, j) < 0$. Then, in $D(S)$, every longest path from p_0 to p_{n+1} contains g_j.

Proof. Since every path from p_0 to p_{n+1} goes through either p_{j+1} or g_{j+1}, it suffices to show that g_j is contained in every longest path from p_0 to p_{j+1} and from p_0 to g_{j+1}. Since $S \in T(g_j)$ (by definition of j), the length of a longest path from p_0 to p_j (resp. g_j) is given by Lemma 5.3. Thus, it suffices to show that

$$(L_P(0, j+1) - s(S, j) - d(g_j, p_{j+1})) + d(g_j, p_{j+1}) > L_P(0, j) + d(p_j, p_{j+1}) \quad (5.6)$$

(i.e., the longest path to p_{j+1} via g_j is longer than the longest path to p_{j+1} via p_j), and that

$$(L_P(0, j+1) - s(S, j) - d(g_j, p_{j+1})) + d(g_j, g_{j+1}) > L_P(0, j) + d(p_j, g_{j+1}) \quad (5.7)$$

(i.e., the longest path to g_{j+1} via g_j is longer than the longest path to g_{j+1} via g_j).

Now, (5.6) is trivially equivalent to the assumption that $s(S, j) < 0$. On the other hand, according to Definition 5.1,

$$d(g_j, g_{j+1}) + d(p_j, p_{j+1}) \geq d(g_j, p_{j+1}) + d(p_j, g_{j+1}). \quad (5.8)$$

The inequality (5.7) is then obtained by the addition of (5.8) to (5.6). ∎

For an arbitrary selection $S = \{g_1, g_2, \ldots, g_n\}$, Lemma 5.4 suggests that a longest path of $D(S)$ can be obtained by the following procedure. (Let us mention right away that this procedure is much more involved than necessary, if its only purpose is to obtain a longest path of $D(S)$. The reason for considering it in this form is that it will rather naturally lead to an algorithm for CRP.) First, compute all labels $s(S, j)$ (e.g. layer by layer, as suggested by Lemma 5.2). If all s-labels are nonnegative, then we know that $(p_0, p_1, \ldots, p_{n+1})$ is a longest path of $D(S)$. Otherwise, let

$$j = \min\{k \in \{1, \ldots, n\} \mid s(S, k) < 0\}.$$

In view of Lemma 5.4, a longest path from p_0 to p_{n+1} in $D(S)$ can be obtained by concatenating a longest path from p_0 to g_j with a longest path from g_j to p_{n+1}. Accordingly, for any selection S, we call the first grip node $g_j \in S$ for which $s(S, j)$ is negative, a *reset node* of $D(S)$. The terminology *reset* expresses the fact that the computation of a longest path of $D(S)$ can be started anew from such a node. Now, by Lemma 5.3, a longest path from p_0 to g_j is readily available. Thus, we only need to find a longest path from g_j to p_{n+1} in $D(S)$. This subproblem clearly has the same structure as the problem we started with. More precisely, we can handle it as follows. We discard from $D(S)$ all layers with index $i \leq j$, except for g_j. Moreover, we decrease the length of both arcs (g_j, g_{j+1}) and (g_j, p_{j+1}) by $d(g_j, p_{j+1})$ (this is to account for the last term of (5.5); see (5.9) hereunder). Denote the new CRP graph thus constructed

by $D_{g_j}(S)$. The observation we make now is that, as a consequence of Lemma 5.3 and Lemma 5.4,

$$L(D(S)) = L_P(0, j+1) - s(S, j) + L(D_{g_j}(S)) \tag{5.9}$$

(cf. Definition 5.3). The procedure just described can be applied iteratively until either g_n receives a nonnegative label or g_n becomes a reset node. In either case, let $u_1, u_2, \ldots, u_r$ denote the reset nodes sequentially identified in the process. Thus, for $k = 1, \ldots, r$, u_k is the reset node of $D_{u_{k-1}}(S)$ (where we let $u_0 \equiv p_0$). Denote by $s(S, u_{k-1}, u_k)$ the (negative) s-label attached to u_k in $D_{u_{k-1}}(S)$. The previous discussion can then be summarized as follows.

Lemma 5.5 If u_k is the reset node of $D_{u_{k-1}}(S)$ for $k = 1, \ldots, r$, and $D_{u_r}(S)$ has no reset node, then

$$L(D(S)) = L_P(0, n+1) - \textstyle\sum_{k=1}^{r} s(S, u_{k-1}, u_k). \tag{5.10}$$

Proof. This statement is a consequence of Lemma 5.3 and Lemma 5.4, and the foregoing discussion. More precisely, let u_r lie in G_ℓ, where $1 \leq \ell \leq n$. Then, induction on (5.9) leads to

$$L(D(S)) = L_P(0, \ell+1) - \textstyle\sum_{k=1}^{r} s(S, u_{k-1}, u_k) + L(D_{u_r}(S)).$$

There are now two cases. If $\ell = n$, i.e. the reset node u_r coincides with $g_n \in S$, then $L(D_{u_r}(S)) = 0$, from which (5.10) follows. Conversely, if g_n is not a reset node, then $L(D_{u_r}(S)) = L_P(\ell+1, n+1)$ (by Corollary 5.1), and (5.10) follows again. ∎

Finally, consider two selections in $T(g_j)$, which are identical from layer j onwards, but which have different s-label values at layer j. The next lemma states sufficient conditions for one of the selections to dominate the other one, as far as minimizing $L(D(S))$ is concerned.

Lemma 5.6 For any $j \in \{1, 2, \ldots, n\}$ and $g_j \in G_j$, let $S = \{g_1, g_2, \ldots, g_n\}$ and $S' = \{g'_1, g'_2, \ldots, g'_n\}$ be two selections in $T(g_j)$ with $g_i = g'_i$ for $i = j, \ldots, n$. If $s(S, j) < s(S', j)$ and $s(S, j) < 0$, then $L(D(S')) < L(D(S))$.

Proof. Let P be any longest path in $D(S')$. Then, P contains either p_j or g_j. In the first case, we can assume without loss of generality that P contains $p_0, \ldots, p_j$ (cf. Lemma 5.3 sub (i)), so that P is also a path in $D(S)$. But then Lemma 5.4 implies that P is not a longest path of $D(S)$, hence $L(D(S')) < L(D(S))$. Conversely, if P contains g_j, then Lemma 5.3 sub (ii) states that the subpath of P from p_0 to g_j has length $L_P(0, j+1) - s(S', j) - d(g_j, p_{j+1})$, and that the longest path from p_0 to g_j in $D(S)$ has length $L_P(0, j+1) - s(S, j) - d(g_j, p_{j+1})$. Since the latter is strictly larger than the former, the result follows. ∎

5.5.3 The general case

Below we are going to show how Lemmas 5.5 and 5.6 can be combined to produce a polynomial time algorithm for CRP. First, we reformulate and extend some of the notation introduced earlier. For every node $g_j \in G_j$ of D ($j = 0, 1, \ldots, n-1$), we denote by D_{g_j} the subgraph of D induced by the node set $\{g_j\} \cup \bigcup_{i=j+1}^{n+1} L_i$. The arc lengths in D_{g_j} are the same as in D, except that the length of each arc leaving g_j is decreased by $d(g_j, p_{j+1})$ for all $j \geq 1$. Note that D_{g_0} is thus identical to D. For each graph D_{g_j}, we can define s^*-labels as we did for graph D in Definition 5.7; the s^*-label attached to node g_k in D_{g_j} is denoted by $s^*(g_j, g_k)$ ($g_k \in \bigcup_{i=j+1}^{n} G_i$). In addition, $S_{g_j g_k}$ will refer to any selection that realizes the value of $s^*(g_j, g_k)$.

Before giving a more formal description of our algorithm, let us clarify the intuition behind it. Observe that, according to (5.10), CRP can be seen as the problem of minimizing the expression $\sum_k -s(S, u_{k-1}, u_k)$ over all possible selections S. Let S be an optimal selection of the CRP graph D, and let $u_1, \ldots, u_r$ be the corresponding sequence of reset nodes. By definition of the quantities $s(S, u_{k-1}, u_k)$, we have $s(S, u_0, u_1) = s(S, j)$ if u_1 is in layer j. Consider now the selection $S_{u_0 u_1}$, which is such that (by definition)

$$s(S_{u_0 u_1}, j) = s^*(u_0, u_1) = \max_{R \in T(u_1)} s(R, j),$$

and suppose that

$$s(S, j) < s(S_{u_0 u_1}, j).$$

Then, construct the selection S' that coincides with $S_{u_0 u_1}$ from layer 1 to layer j and that coincides with S from layer j to layer n. It follows directly from Lemma 5.6 that S' dominates S, and this contradicts the optimality of S. Thus we have established that

$$s(S, j) = s^*(u_0, u_1),$$

or equivalently

$$s(S, u_0, u_1) = s^*(u_0, u_1).$$

By repeating this argument r times, we can derive that

$$-\sum_{k=1}^{r} s(S, u_{k-1}, u_k) = -\sum_{k=1}^{r} s^*(u_{k-1}, u_k). \tag{5.11}$$

In this way, we have reduced CRP to the problem of minimizing the right-hand side of (5.11) over all possible choices of the u_k's, under the restriction that these nodes are the sequence of reset nodes associated with some selection.

We will now translate the latter problem into a shortest path problem in an auxiliary network $D_{\mathcal{N}}$, where the length of each arc (g_j, g_k) is "essentially" equal to $-s^*(g_j, g_k)$. More precisely, the node set of $D_{\mathcal{N}}$ is $\{g_0\} \cup \bigcup_{i=1}^{n} G_i$. The arcs of $D_{\mathcal{N}}$ are all pairs of

nodes of the form (g_j, g_k) where $g_j \in G_j$, $g_k \in G_k$ and $0 \leq j < k \leq n$. The length of arc (g_j, g_k) is defined to be $w(g_j, g_k)$, where

$$\text{Case 1: } w(g_j, g_k) = -s^*(g_j, g_k) \text{ if } -\infty < s^*(g_j, g_k) < 0; \tag{5.12}$$

$$\text{Case 2: } w(g_j, g_k) = 0 \text{ if } s^*(g_j, g_k) \geq 0 \text{ and } k = n; \tag{5.13}$$

$$\text{Case 3: } w(g_j, g_k) = \infty \text{ otherwise.} \tag{5.14}$$

In view of Definition 5.7 and Theorem 5.3, Case 1 corresponds to a situation where g_k is a reset node in the subgraph of D_{g_j} induced by the selection $S_{g_j g_k}$ (see property P1 following Definition 5.7). Similarly, Case 2 occurs when there is no reset node in the subgraph induced by $S_{g_j g_n}$ up to and including layer n. Finally, in Case 3, any reset node of the subgraph induced by $S_{g_j g_k}$ lies either before g_k ($s^*(g_j, g_k) = -\infty$) or after g_k ($s^*(g_j, g_k) \geq 0$ and $k < n$).

Denote by $w(P)$ the length of a path P in $D_{\mathcal{N}}$. For brevity, when we write "shortest path in $D_{\mathcal{N}}$", we mean "shortest path in $D_{\mathcal{N}}$ from g_0 to some node in G_n, with respect to the length function w". We are now finally ready for our next, and main, result. Note, however, that its proof is simply a formal generalization of the arguments presented above.

Theorem 5.4 The length of a shortest path in $D_{\mathcal{N}}$ is equal to $L_P(0, n+1) + L(D)$, where $L(D)$ is the optimal value of the Component Retrieval Problem on the graph D.

Proof.

(i). Let S be an optimal selection for CRP and let u_k denote the reset node of $D_{u_{k-1}}(S)$ ($k = 1, \ldots, r$; $u_0 = g_0$). By Lemma 5.5, equation (5.10) holds. Let now $P' = \{u_0, u_1, \ldots, u_r\}$, and consider any index $k \in \{1, \ldots, r\}$. By definition of reset nodes, $s(S, u_{k-1}, u_k) < 0$ and $s(S, u_{k-1}, u) \geq 0$ for all grip nodes u lying between u_{k-1} and u_k in S. Therefore, by Definition 5.7, we get $S \in T(u_k)$, where $T(u_k)$ is defined with respect to the graph $D_{u_{k-1}}$, and this implies that $-\infty < s(S, u_{k-1}, u_k) \leq \min\{0, s^*(u_{k-1}, u_k)\}$. If, for all $k = 1, \ldots, r$, $w(u_{k-1}, u_k)$ is defined by either Case 1 or Case 2 (cf. (5.12)-(5.13)), then,

$$-s(S, u_{k-1}, u_k) \geq w(u_{k-1}, u_k) \quad \text{for } k = 1, \ldots, r.$$

Combining these inequalities with (5.10) yields

$$L(D) \geq L_P(0, n+1) + w(P'). \tag{5.15}$$

Thus, assume now that $w(u_{k-1}, u_k)$ is defined by Case 3 (see (5.14)) for some k. In that case, $s^*(u_{k-1}, u_k) \geq 0$, or equivalently $s(S_{u_{k-1}u_k}, u_{k-1}, u_k) \geq 0$ (notice that $s^*(u_{k-1}, u_k) = -\infty$ has been ruled out earlier). Let now S' denote the selection which coincides with S from p_0 to u_{k-1} and from u_k to p_{n+1}, and which coincides with $S_{u_{k-1}u_k}$ from u_{k-1} to u_k. Apply Lemma 5.6 to the selections S and S',

both viewed as selections of $D_{u_{k-1}}$. This lemma implies that the longest path in the subgraph of $D_{u_{k-1}}$ induced by S' is shorter than the longest path in the subgraph induced by S, contradicting the optimality of S. As a result, the case that $w(u_{k-1}, u_k)$ is defined by (5.14) does not occur for arcs (u_{k-1}, u_k) on paths in $D_{\mathcal{N}}$ that are defined by the reset nodes of optimal selections.

(ii). Conversely, let $P = \{u_0, u_1, \ldots, u_r\}$ be a shortest path in $D_{\mathcal{N}}$, with $u_0 = g_0$ and $u_r \in G_n$. From (i) it follows that $w(P) \leq w(P') < \infty$. Hence, for $k = 1, \ldots, r-1$, the $w(u_{k-1}, u_k)$'s on P are all defined by (5.12), which means that u_k is a reset node in the subgraph of $D_{u_{k-1}}$ induced by the selection $S_{u_{k-1}u_k}$. Now consider the selection $S = \cup_{1 \leq k \leq r} S'_{u_{k-1}u_k}$, where $S'_{u_{k-1}u_k}$ is the set of nodes of $S_{u_{k-1}u_k}$ that lie between u_{k-1} and u_k. Lemma 5.5 implies that $L(D(S)) = L_P(0, n+1) + w(P)$, and hence

$$L(D) \leq L_P(0, n+1) + w(P). \tag{5.16}$$

From (5.15) and (5.16), we conclude that $L(D) \leq L_P(0, n+1) + w(P) \leq L_P(0, n+1) + w(P') \leq L(D)$. This establishes the result. ∎

In summary, the Component Retrieval Problem can be solved by the algorithm of Figure 5.4.

Theorem 5.5 Procedure **SOLVE-CRP** is correct and solves the Component Retrieval Problem in $O(ve)$ time on a CRP graph D with v nodes and e arcs.

Proof. The correctness of procedure **SOLVE-CRP** follows from (the proof of) Theorem 5.4. As for its complexity, note that each execution of the loop "**for all** j, **for all** g_j" requires $O(e)$ time (by the comments following Theorem 5.3), and that this loop is executed $O(v)$ times. A shortest path in $D_{\mathcal{N}}$ can be found in $O(v^2)$ time since $D_{\mathcal{N}}$ is acyclic (see e.g. Ahuja, Magnanti and Orlin [1993]). ∎

The complexity of procedure **SOLVE-CRP** can be alternatively stated as follows. Let m be an upper-bound on the number of feeders of each type, i.e. $m = \max_{1 \leq i \leq n} |G_i|$. Then, $v = O(mn)$ and $e = O(m^2 n)$, so that **SOLVE-CRP** runs in $O(m^3 n^2)$ time.

5.5.4 Example

Below we will illustrate the algorithm of the previous subsection by applying it to the problem instance that was considered in Section 5.3 with $n = 5$. Recall that $d(g_4, g_5) = 4 + x$ in this problem, with x being equal to 0 and 5 respectively. The first phase yields the s^*-labels; the relevant ones are listed in Table 5.4.

The auxiliary graph $D_{\mathcal{N}}$ has node set $\{g_0, g_1^1, g_1^2, g_2^1, g_2^2, g_3, g_4, g_5\}$; its relevant arcs are listed in Table 5.5. If $x = 0$, the shortest path in $D_{\mathcal{N}}$ is (g_0, g_3, g_5) with a length

```
procedure SOLVE-CRP:

begin

    for all j = 0, 1, ..., n − 1 and for all g_j ∈ G_j do
    begin
        set up the graph D_{g_j};
        for all k = j + 1, ..., n and all g_k ∈ G_k do
        begin
            compute the label s*(g_j, g_k) of node g_k in D_{g_j}, and the corresponding
              selection S_{g_j g_k};
            define w(g_j, g_k) according to (5.12), (5.13) and (5.14);
        end
    end
    set up the graph N;
    compute a shortest path in D_N from g_0 to G_n with respect to the length function
      w;
    let P = {u_0, u_1, ..., u_r} denote this shortest path;
    return the optimal selection S = ∪_{1≤k≤r} S'_{u_{k-1} u_k} with length L(D) = L_P(0, n +
      1) + w(P)

end
```

Figure 5.4: Algorithm for solving CRP

grip node pair (g_i, g_j)	$s^*(g_i, g_j)$	arg max in (5.5)
(g_0, g_1^1)	2	g_0
(g_0, g_1^2)	2	g_0
(g_0, g_2^1)	2	g_1^1
(g_0, g_2^2)	-4	g_1^2
(g_0, g_3)	-2	g_2^1
(g_2^2, g_3)	3	g_2^2
(g_2^2, g_4)	3	g_3
(g_3, g_4)	0	g_3
(g_2^2, g_5)	$3 - x$	g_4
(g_3, g_5)	$-x$	g_4

Table 5.4: (Relevant) s^*-labels of the example CRP problem.

Arc	Length	Remark
(g_0, g_2^2)	4	
(g_0, g_3)	2	
(g_2^2, g_5)	2	only if $x = 5$
(g_3, g_5)	5	only if $x = 5$
(g_2^2, g_5)	0	only if $x = 0$
(g_3, g_5)	0	only if $x = 0$

Table 5.5: (Relevant part of) graph $D_{\mathcal{N}}$ of the example CRP.

of 2. Tracing back the predecessors in the third column of Table 5.4 reveals the corresponding optimal selection $\{g_1^1, g_2^1, g_3, g_4, g_5\}$, which has a makespan of 16+2=18 (cf. Theorem 5.4). On the other hand, if $x = 5$, then the shortest path in $D_{\mathcal{N}}$ is (g_0, g_2^2, g_5) with a length of 6. The corresponding optimal selection reads $\{g_1^2, g_2^2, g_3, g_4, g_5\}$, which has a makespan of 16+6=22. Note that these outcomes are consistent with the optimal selections that were reported in Section 5.4.

5.6 An *NP*-hard generalization of CRP.

Our definition of the Component Retrieval Problem includes condition (5.1) on the arc lengths of CRP graphs. This condition has been explicitly used in the proof of Lemma 5.5 in Section 5.5. In this section we will show that the problem becomes *NP*-hard when condition (5.1) is absent. Consider the following decision problem (*Generalized CRP*).

Definition 5.8 GCRP

INPUT: An integer β and a graph D satisfying the assumptions of Definition 5.1, except for (5.1).

QUESTION: Is there a selection S such that the longest path in the selection induced subgraph $D(S)$ has length at most β?

Theorem 5.6 GCRP is *NP*-complete, even if $|G_i| \leq 2$ for $i = 1, \ldots, n$.

Proof. GCRP is clearly in *NP*. Below we will present a polynomial transformation from the *NP*-complete *Even-Odd Partitioning* problem (EOP; see Garey and Johnson [1979]) to CRP.

Definition 5.9 EOP

INPUT: N pairs of positive integers $I_i = \{x_{2i-1}, x_{2i}\}$ for $i = 1, \ldots, N$.

QUESTION: Is there an even-odd partition of $\{1, 2, \ldots, 2N\}$, i.e. a partition of $\{1, 2, \ldots, 2N\}$ into disjoint subsets A and B with $|A \cap I_i| = |B \cap I_i| = 1$ for $i = 1, \ldots, N$, and $\sum_{i \in A} x_i = \sum_{i \in B} x_i$?

Given an instance of EOP, we define a graph D as in Definition 5.1, with $n = 4N$. For $k = 1, \ldots, 4N$, each layer k contains three nodes, namely one place node p_k and two grip nodes g_k^1 and g_k^2. Layers $4i-3, 4i-2, 4i-1$ and $4i$ are associated with pair I_i in the instance of EOP ($1 \leq i \leq N$). In order to define the arc lengths, we introduce three large numbers of different magnitudes:

$$\begin{aligned} Q &= (N+1) \cdot \max_{1 \leq i \leq 2N} \{x_i\} \\ K &= (N+1)Q \\ M &= 4(N+1)K \end{aligned}$$

The arc lengths in D are listed in Table 5.6. In this table, we call an arc connecting two place nodes a place arc, and an arc connecting a grip and a place node, a cross arc. (For notational convenience we indifferently denote the sink of D by p_{n+1}, g_{n+1}^1 or g_{n+1}^2.)

Recall that arcs emanating from g_0 have length zero. Finally, we set $\beta = N(3K + Q) + \frac{1}{2}\sum_{i=1}^{2N} x_i$. This completely specifies an instance (D, β) of GCRP. It remains to show that the instance of GCRP obtained in this way and the original instance of EOP have the same answer.

Arc	Length	For
(g^1_{4i-3}, g^1_{4i-2})	$K + x_{2i-1}$	$i = 1, \ldots, N$
(g^2_{4i-3}, g^2_{4i-2})	$K + x_{2i}$	$i = 1, \ldots, N$
(g^1_{4i-2}, g^1_{4i-1})	0	$i = 1, \ldots, N$
(g^2_{4i-2}, g^2_{4i-1})	0	$i = 1, \ldots, N$
(g^1_{4i-1}, g^1_{4i})	$K + x_{2i}$	$i = 1, \ldots, N$
(g^2_{4i-1}, g^2_{4i})	$K + x_{2i-1}$	$i = 1, \ldots, N$
(g^1_{4i}, g^1_{4i+1})	K	$i = 1, \ldots, N$
(g^1_{4i}, g^2_{4i+1})	K	$i = 1, \ldots, N$
(g^2_{4i}, g^1_{4i+1})	K	$i = 1, \ldots, N$
(g^2_{4i}, g^2_{4i+1})	K	$i = 1, \ldots, N$
(g^1_k, g^2_{k+1})	M	$k \in \{4i-3, 4i-2, 4i-1\}$
(g^2_k, g^1_{k+1})	M	$k \in \{4i-3, 4i-2, 4i-1\}$
(p_{4i-2}, p_{4i-1})	0	$i = 1, \ldots, N$
other place arcs	K	
all cross arcs	Q	

Table 5.6: Arc lengths of the GCRP instance (D, β)

(i). Suppose first that the instance of EOP has a positive answer, and let (A, B) define an even-odd partition of $\{1, 2, \ldots, 2N\}$. Without loss of generality we may assume that $A = \{2i-1 | i = 1, \ldots, N\}$ and $B = \{2i | i = 1, \ldots, N\}$. Consider the selection S that contains g^1_{4i-3}, g^1_{4i-2}, g^1_{4i-1} and g^1_{4i} for i is odd, and g^2_{4i-3}, g^2_{4i-2}, g^2_{4i-1} and g^2_{4i} for i is even $(i = 1, \ldots, N)$. We now claim that $L(D(S)) = \beta$, implying that the instance (D, β) has a positive answer.

Denote by D_i the subgraph of $D(S)$ induced by layers $4i-3, 4i-2, \ldots, 4i+1$, for every $i \in \{1, 2, \ldots, N\}$. Two candidate longest paths in D_i are of the form

$$\{p_{4i-3}, p_{4i-2}, g^{\delta}_{4i-1}, g^{\delta}_{4i}, g^{3-\delta}_{4i+1}\} \text{ and } \{g^{\delta}_{4i-3}, g^{\delta}_{4i-2}, p_{4i-1}, p_{4i}, p_{4i+1}\}, \tag{5.17}$$

respectively, where $\delta = 1$ when i is odd, and $\delta = 2$ otherwise. One of these paths has length $3K + Q + x_{2i-1}$ and the other one has length $3K + Q + x_{2i}$. Furthermore, it is easily seen that all other paths in D_i are strictly shorter than the ones in (5.17). Using mathematical induction to N then reveals that any longest path from g_0 to p_{4N+1} in $D(S)$ is the concatenation of paths in D_i of the types mentioned in (5.17) $(i = 1, 2, \ldots, N)$. Hence, the two candidate longest paths in $D(S)$ are

$$P_1 = (g_0, g^1_1, g^1_2, p_3, p_4, p_5, p_6, g^2_7, g^2_8, g^1_9, g^1_{10}, p_{11}, p_{12}, p_{13}, p_{14}, g^2_{15}, g^2_{16}, g^1_{17}, \ldots, p_{4N+1})$$

$$= \{g_0\} \cup \bigcup_{\substack{i=1, \\ i \text{ odd}}}^{N} \{g^1_{4i-3}, g^1_{4i-2}, p_{4i-1}, p_{4i}\} \cup \bigcup_{\substack{i=1 \\ i \text{ even}}}^{N} \{p_{4i-3}, p_{4i-2}, g^2_{4i-1}, g^2_{4i}\} \cup \{p_{4N+1}\}$$

and

$$P_2 = (g_0, p_1, p_2, g_3^1, g_4^1, g_5^2, g_6^2, p_7, p_8, p_9, p_{10}, g_{11}^1, g_{12}^1, g_{13}^2, g_{14}^2, p_{15}, p_{16}, p_{17}, \ldots, p_{4N+1})$$
$$= \{g_0\} \cup \bigcup_{\substack{i=1,\\ i \text{ odd}}}^{N} \{p_{4i-3}, p_{4i-2}, g_{4i-1}^1, g_{4i}^1\} \cup \bigcup_{\substack{i=1,\\ i \text{ even}}}^{N} \{g_{4i-3}^2, g_{4i-2}^2, p_{4i-1}, p_{4i}\} \cup \{p_{4N+1}\}.$$

with lengths $N(3K+Q) + \sum_{i=1}^{N} x_{2i-1} = N(3K+Q) + \sum_{i \in A} x_i = \beta$ and $N(3K+Q) + \sum_{i \in B} x_i = \beta$ respectively. Consequently, both candidate longest paths are in fact longest paths, and the answer to the GCRP instance (D, β) is like the answer to the EOP instance, viz. affirmative.

(ii). Suppose next that the answer to the GCRP instance (D, β) is affirmative, and let S be a selection of D with $L(D(S)) \leq \beta$. Since M is very large, $D(S)$ cannot contain any arc with length M. This means that, for each quadruple of layers $4i-3, 4i-2, 4i-1$ and $4i$ $(i = 1, 2, \ldots, N)$, either all four grip nodes g_{4i-3}^1, g_{4i-2}^1, g_{4i-1}^1 and g_{4i}^1, or all four grip nodes g_{4i-3}^2, g_{4i-2}^2, g_{4i-1}^2 and g_{4i}^2 are in S. Therefore, S can be denoted by

$$S = \{g_0\} \cup \bigcup_{i=1}^{N} \{g_{4i-3}^{\alpha_i}, g_{4i-2}^{\alpha_i}, g_{4i-1}^{\alpha_i}, g_{4i}^{\alpha_i}\} \cup \{p_{4N+1}\}$$

with $\alpha_i \in \{1, 2\}$ $(i = 1, \ldots, N)$. Now consider the two paths

$$P_1' = (g_0, g_1^{\alpha_1}, g_2^{\alpha_1}, p_3, p_4, p_5, p_6, g_7^{\alpha_2}, g_8^{\alpha_2}, g_9^{\alpha_3}, g_{10}^{\alpha_3}, p_{11}, p_{12}, p_{13}, p_{14}, g_{15}^{\alpha_4}, g_{16}^{\alpha_4}, g_{17}^{\alpha_5}, \ldots, \ldots, p_{4N+1})$$
$$P_2' = (g_0, p_1, p_2, g_3^{\alpha_1}, g_4^{\alpha_1}, g_5^{\alpha_2}, g_6^{\alpha_2}, p_7, p_8, p_9, p_{10}, g_{11}^{\alpha_3}, g_{12}^{\alpha_3}, g_{13}^{\alpha_4}, g_{14}^{\alpha_4}, p_{15}, p_{16}, p_{17}, \ldots, \ldots, p_{4N+1})$$

The lengths of these paths are $L(P_1') = N(3K+Q) + \sum_{i \in A} x_i$ and $L(P_2') = N(3K+Q) + \sum_{i \in B} x_i$ respectively, where (A, B) is a partition of $\{1, \ldots, 2N\}$ with $|A \cap I_i| = |B \cap I_i| = 1$ for $i = 1, \ldots, N$. Since both P_1' and P_2' have lengths that are at most the longest path length in $D(S)$, and since the latter on its turn is at most β, it follows that $L(P_1') \leq \beta$ and $L(P_2') \leq \beta$. These observations, combined with the choice of $\beta = N(3K+Q) + \frac{1}{2}\sum_{i=1}^{2N} x_i$ renders $\sum_{i \in A} x_i = \sum_{i \in B} x_i$. Hence (A, B) is an even-odd partition of $\{1, \ldots, 2N\}$, thus establishing the fact that like the GCRP instance (D, β), the EOP instance allows for an affirmative answer as well. ∎

As a final comment it may be worth noticing that, as expected, the instance of GCRP that is created in the proof of Theorem 5.6, does *not* satisfy condition (5.1) (thereby leaving the $P = NP$ question unanswered). Indeed, for $i = 1, 2, \ldots, N$, we have

$$d(g_{4i-2}^1, g_{4i-1}^1) + d(p_{4i-2}, p_{4i-1}) = 0 < 2Q = d(g_{4i-2}^1, p_{4i-1}) + d(p_{4i-2}, g_{4i-1}^1),$$

which contradicts condition (5.1).

5.7 Conclusions

The main contribution of this chapter is a "two-phase", polynomial time dynamic programming algorithm for the Component Retrieval Problem, a problem that arises in the automated assembly of printed circuit boards. We have broadened the scope of our analysis by modelling the problem as a longest path minimization problem in a PERT/CPM-like network with design aspects. As an alternative interpretation, the problem can also be viewed as a shortest path problem with side-constraints. Both interpretations have proven to be crucial in the development and description of the proposed solution algorithm. Finally, we have sharply delineated the complexity of the problem by proving that it becomes *NP*-hard when additional structure on the activity durations in the PERT/CPM network is absent.

Chapter 6

A case study in printed circuit board assembly

6.1 Introduction

The assembly of printed circuit boards (PCBs) is in general a complicated task. Sophisticated machines must perform intricate operations, involving different kinds of tools and various components, in order to assemble a board. Numerous constraints and conflicting objectives interfere to create a challenging planning problem. Further, the competition faced by a PCB-manufacturer causes a need for high throughput rates. In order to cope with such an environment, it is nowadays well recognized that the availability of automated planning procedures is a major asset.

In this chapter we describe a typical case in PCB-assembly arising at a plant of Philips NV, a major PCB-manufacturer. We propose a planning procedure for a situation where a family of different board types are to be produced by a line of different placement machines. This procedure is based on a hierarchical decomposition of the planning problem. Not surprisingly, most subproblems in this decomposition are already very hard in terms of computational complexity. Moreover, the size of the problems we consider prohibits the effective use of exact solution methods. In order to obtain good solutions in a reasonable amount of time, we solve the subproblems approximately using heuristics and local search methods. The resulting procedure is tested on real-life instances (made available to us by Philips), for which we are able to close about 70% of the gap between the previously best-known makespan and a (fairly) weak lower bound, thus reducing the overall makespan by approximately 17%.

Let us now give a short description of how the assembly of PCBs is organized at the Philips plant under study. There are a number of assembly-lines, each consisting of a number of placement machines which place electronic components on bare boards. There are different types of boards and different types of components to consider. Each board type is assigned to a line of placement machines, that is, all boards of a type 'flow' through the machines of a specific line. The electronic components must be mounted at prescribed locations on a board. The set of locations to be served on a

board as well as the type of components to be placed at each location depends upon the board type. In other words, for each board type, a set of locations, and for each of these locations the type of the component to be placed there, is given. Components of each type are delivered to the machine by means of a so-called feeder. Each placement machine is equipped with a feeder rack which holds the feeders. Further, the machine has some device - dependent upon the technology (see Section 6.2) - which is able to pick components from the feeders and place these components onto the board.

A production plan associated with this description should at least specify the following:

(1). a *partition* of the set of board types into *families* which are to be assigned to different lines of placement machines,

(2). for each board type, a *partition* of the set of component locations on this board, that is a decision concerning which locations are going to be served by which machine,

(3). for each machine, a *feeder rack assignment*, that is an assignment of feeders to positions in the feeder rack,

(4). for each pair consisting of a machine and a board type, a *component placement sequence*, that is an order in which components are placed at the locations on this board that are served by this machine, and

(5). for each pair consisting of a machine and a board type, a *component retrieval plan*, that is a decision for each component from which feeder it is to be retrieved.

In this chapter, we focus on problems (2)-(5), thus we deal with planning problems that arise when a given family of board types is assembled by a single line of placement machines. In Section 6.2 we give a precise description of these problems (including some of the technological features of the placement machines used), and in Section 6.3 we describe our solution procedure. Section 6.4 is devoted to the performance of our procedure on real-life instances and Section 6.5 contains the conclusions. The remainder of the current section is devoted to literature related to the assembly of PCBs.

In case one strives to minimize the makespan for a single machine producing a single board type, the planning problem above reduces to problems (3), (4) and (5) (feeder rack assignment, component placement sequence and component retrieval plan). A number of studies focus on problems (3) and (4) only, since problem (5) vanishes when precisely one feeder is available per component type. In such a case, the interaction between problems (3) and (4) is of crucial importance. This issue has been identified by Drezner and Nof [1984] (see also Walas and Askin [1984] for a similar application in the production of metal parts). An approach based on location theory is reported by Foulds and Hamacher [1993]. Leipälä and Nevalainen [1989] suggest a solution procedure based on heuristically solving a TSP and a quadratic assignment problem in an

iterative fashion. Other heuristic approaches are described in Francis, Hamacher, Lee and Yeralan [1994] (see also Viczián [1993]) and Younis and Cavalier [1990]. Ball and Magazine [1988] show that, when a feeder rack assignment is given, an optimal component placement sequence can, in some situations, be found in polynomial time. Bard, Clayton and Feo [1994] propose a planning procedure tailored for a specific placement machine, in which problem (5) is explicitly addressed. (see also Crama, Flippo, van de Klundert and Spieksma [1995a], and Chapter 5). Further, Ahmadi, Grotzinger and Johnson [1988] present a model which determines, among other parameters, with how many feeders of each type the placement machine should be equipped.

When the planning problem is extended to a line of placement machines producing a single board type, problem (2) comes up. Crama, Kolen, Oerlemans and Spieksma [1994] and van Laarhoven and Zijm [1993] each propose a solution procedure for a line of CSM-60 placement machines. Lofgren, McGinnis and Tovey[1991] treat a case where a board is allowed to visit a machine more than once.

Relatively few published studies deal with the case where multiple boards types are to be assembled by one or more machines: we mention Carmon, Maimon and Dar-el [1989], Balakrishnan and Vanderbeck [1993] and Askin, Dror and Vakharia [1994]. In this situation the following issues may appear (cf. problem (1)). Since the set of component types needed to produce all board types can be larger than the available capacity in the feeder racks, one may be forced to partition the set of board types into subfamilies which can be produced with a fixed feeder assignment. Thus, one may have to solve problems (2)-(5) a number of times during the planning period to produce all board types. This problem is addressed in Tang and Denardo [1988b], and Bard [1988]. A number of authors have further investigated this *batch selection* problem (see Crama [1995] for further references). Alternatively, given a number of assembly-lines, one may try to solve problem (1) in such a way that each family can be produced on a line (which describes the situation at hand, see Section 6.2). Also, one may partly circumvent the problem by prescribing some feeders (the 'common' feeders) to be permanently assigned to certain positions in the rack, whereas other feeders (the 'exotic' ones) are loaded as needed, see Carmon, Maimon and Dar-el [1989], Balakrishnan and Vanderbeck [1993] and Agnetis, Askin and Sodhi [1994] for a description of this idea. Askin, Dror and Vakharia [1994] study the assembly of multiple board types by multiple machines in an open shop (rather than flowshop) environment. Another issue which becomes apparent is that, when dealing with more than one board type, problem (3) becomes much more complicated. Indeed, most of the planning procedures described in the literature we mentioned here, attempt to determine a feeder rack assignment for which an optimal component placement sequence can be found. However, when dealing with multiple board types, one has to find a single feeder rack assignment such that a good placement sequence can be found *for each* board type (see sections 2 and 3).

Finally, the technology employed by the placement machine under consideration may give rise to specific planning problems (see for instance Ahmadi and Kouvelis [1994]). An overview of issues which arise in the operational planning of PCB assembly can be found in Ahmadi [1993], Crama, Oerlemans and Spieksma [1994] and Voogt

[1993].

6.2 Problem description

In this section we focus on a precise description of problems (2)-(5) for the situation encountered at a plant of Philips. Subsection 6.2.1 refines the description of the setting given in the introduction. Subsection 6.2.2 is devoted to the technological features of the placement machines under consideration. In Subsection 6.2.3 we show that these features allow for an efficient solution of the component retrieval problem, that is problem (5).

6.2.1 Properties of the assembly environment

This subsection deals with the following issues:

- we motivate the choice of our objective function,
- we discuss the nature of the feeder rack assignment problem for our situation, and
- we describe the so-called component retrieval problem.

At the plant investigated, several lines of placement machine are devoted to the assembly of PCBs. Production is mostly organized in such a way that changing feeders for other purposes than refilling should not occur. Thus, the capacity of the feeder racks in the assembly-line restricts the set of board types which the line can handle: this available capacity should be large enough to accommodate all feeders needed to assemble the family of board types assigned to that line. Accordingly, at these plants, problem (1) mentioned in Section 6.1 is reformulated to take this restriction into account. From now on we assume that problem (1) has been solved; thus, we deal with a set of different types of PCBs (a family) that has to be produced by a line of several placement machines (sometimes referred to as the *flowshop*) without any feeder changes.

It is customary at the plant investigated to produce a batch of several hundreds of identical PCBs consecutively, and then switch to another board type. Due to competition and efficiency incentives, it is important to achieve high throughput rates for these batches. Obviously, the throughput rate of each batch is determined by a machine in the line on which the makespan of this board is maximal, called the *bottleneck* machine. Therefore we choose as objective to minimize the sum over the board types of the makespans of these board types on their bottleneck machines. Of course, different types of PCBs, and therefore different batches, may have different bottleneck machines. Thus, more formally, let $t_m^p(s)$ denote the makespan of a board of type p on

machine m for some given solution (i.e. production plan) s. With S denoting the set of feasible solutions, our objective function may be specified as:

$$\min_{s \in S} \sum_{p} \max_{m} t_m^p(s).$$

This seems a reasonable objective function when the batches are of approximately equal size. Otherwise, weights reflecting the size of the batches can be incorporated to obtain a more realistic objective function.

Consider now the feeder rack assignment problem. As mentioned in the introduction, the feeder rack assignment problem becomes harder when multiple board types are involved. In fact, as far as we know, the feeder rack assignment problem with multiple board types has not been addressed explicitly in the literature. Of course, a straightforward way to deal with this problem is to reduce the multiple board case to the single board case. This could be done by creating a so-called *composite* board type which would consist of all the individual board types superposed on each other. In other words, a fictitious board type is made on which all locations of all board types of the family occur. Next, one can apply any existing software for solving the traditional feeder rack assignment problem for this composite board. In fact, this strategy is currently used in practice. Our approach takes a different point of view. A main characteristic of our solution procedure is to take into account as much as possible the *individual* board characteristics. This approach can be motivated by observing that although the set of component types needed for different board types in the family may be similar, the distribution of the locations to be served on those board types can be quite different. Our solution procedure, which is also based on machine characteristics to be discussed later, is described in Section 6.3.

Finally, consider the following issue. Imagine, for reasons of simplicity, the problem of minimizing the makespan of a single board on a single machine. Obviously, a solution to the resulting planning problem must consist of a component placement sequence and a feeder rack assignment. However, in case it is allowed to assign some component type to more than one feeder, solving these two subproblems is not sufficient. In addition, we have to decide for each component *from which feeder* it is to be retrieved. Of course, different decisions for a specific component may result in different makespans for the board. This issue is raised in Bard, Clayton and Feo [1994], and the resulting *component retrieval problem* is further investigated in Crama, Flippo, van de Klundert and Spieksma [1995a] and Chapter 5. Let us refer to an assignment of each location to the feeder delivering the component for this location, as a *component retrieval plan*. Then, the component retrieval problem can be stated as follows:

Given a component placement sequence and a feeder rack assignment, what component retrieval plan minimizes the makespan of the PCB?

Now, the way in which we are able to solve the component retrieval problem (and, in fact some other issues) depends to some extent on the technological features of

the placement machines used. We describe these machines in more detail in the next subsection.

6.2.2 The placement machine

In this subsection we describe the Fuji CP-IV placement machine that is used at the plant investigated. This description will enable us to translate the component retrieval problem for a Fuji CP-IV into a graph-theoretical problem (see Subsection 6.2.3).

Obviously, the task of any PCB assembly machine is to *place* components on a PCB. These components are to be retrieved or *gripped* from feeders. Apart from differences in techniques for gripping and placing (insertion, onsertion, glueing), placement machines differ in the way the coordination between placing and gripping activities is achieved. At the plants we consider, two types of placements (onsertion) machines are used, Fuji CP-III's and Fuji CP-IV's. For our purpose, it is sufficient to assume here that the Fuji CP-III operates in an identical fashion as the Fuji CP-IV, but at a different speed.

To perform its task, the CP-IV is equipped with a gripper, a placer and a carousel. The gripping of components from a feeder is performed by the gripper, and the placement is performed by the placer. The gripper as well as the placer do not move. Each time a component is gripped (or placed), the feeder rack (or the table containing the PCB) is positioned accordingly beneath the gripper (placer). The coordination of the interaction between the gripper and placer is achieved through the carousel or CAM. The carousel has 12 positions configured in a circle and it rotates clockwise in small shifts such that after 12 shifts it has rotated 360 degrees. See Figure 6.1 for a schematic representation of the CP-IV.

Let us now describe how the machine operates. Between two consecutive shifts a component is gripped and a component is placed simultaneously. So, after say $i-1$ shifts ($i \geq 7$), two things happen at the same time: the i-th component to be gripped is gripped and stored in position i mod 12 of the carousel, and the $i-6$-th component to be placed is placed in the PCB from position $(i-6)$ mod 12 of the carousel. Observe that the gripper is six components ahead of the placer. Since the gripping of component i and the placement of component $i-6$ start (and end) at the same time, this implies that, as soon as the grip and the place activities have been performed, three devices start to move at the same time: the table moves to bring the next location beneath the placer, the rack moves to bring the next feeder beneath the gripper and the carousel shifts. The movement which takes maximum time determines the moment when a new iteration starts.

Notice that the modus operandi of the Fuji CP-IV described here (which was communicated to us by Philips) differs slightly from the one given in Bard, Clayton and Feo [1994]. In our situation, the start of a grip activity coincides with the start of a place activity (at least after the first 6 shifts). This requirement is not present in the description given in Bard, Clayton and Feo [1994]. It turns out that this requirement allows us to model the component retrieval problem straightforwardly as a shortest path problem (see Subsection 6.2.3), whereas in the absence of this requirement this is

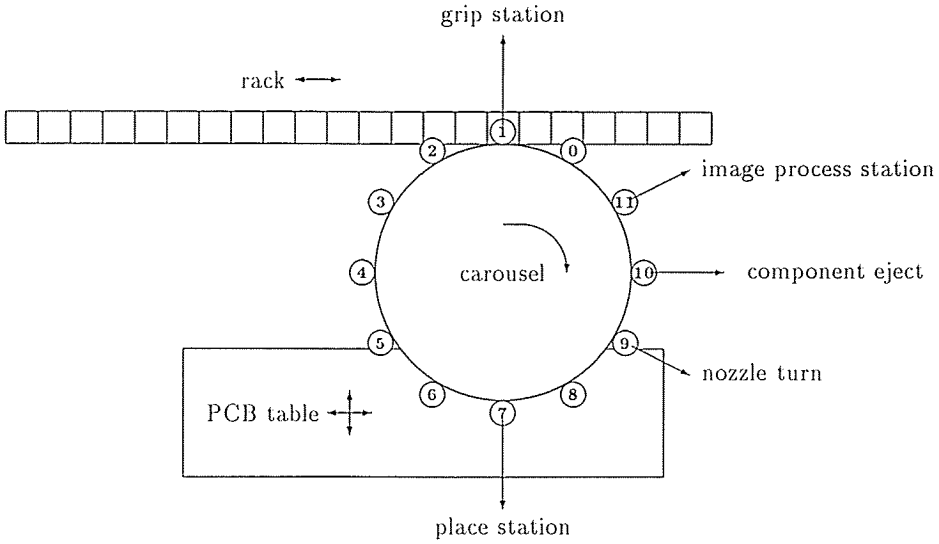

Figure 6.1: The Fuji CP II.

no longer the case (cf. Crama, Flippo, van de Klundert and Spieksma [1995a], Chapter 5).

As a final remark to this subsection, the reader will understand that the description above, although perhaps detailed, is not an exact replica of the truth. For instance, due to the fact that the speed of the carousel depends on the type of the components it carries, not all carousel shifts last equally long. Also, components which have to be rotated may take some extra time of the placer. Further, there are differences between a Fuji CP-III and a Fuji CP-IV placement machine besides speed. Although it is possible to model the aforementioned and other physical characteristics, we have chosen not to do so, for reasons of simplicity and since most of these characteristics will have only marginal effects on the makespans.

6.2.3 The component retrieval problem

How does the description of the placement machine affect the component retrieval problem?

In order to answer this question, recall that, when solving the component retrieval problem, we assume that a component placement sequence and a feeder rack assignment is given. We are going to construct a graph G and show that the component retrieval problem is equivalent to finding a shortest path in this graph.

We assume that the components are placed in numerical order (so component i refers to the i-th component in the component placement sequence). Furthermore, we assume that following the gripping of the last component, six more components are to

be gripped, from the same feeder from which the last component was gripped. These additional 'dummy' grippings are performed in parallel with the last six places, and do not increase the makespan since the feeder rack need not be repositioned to perform them. Similarly we assume that there are six dummy place activities to accompany the first six grip activities, that also do not increase the makespan. So, between each two consecutive carousel rotations, both a grip and a place operation must take place. We define n to be the number of such pairs. (Notice that n exceeds the number of components by six.) Further, let K equal the number of feeder rack positions.

For ease of exposition, we refer to the location on the board corresponding to component i as location i, $1 \leq i \leq n$. Let

Δ gp = time needed to grip (and hence place) a component,
Δc = time needed for a carousel shift,
$\Delta t(i, i+1)$ = time needed for a table movement to bring location $i+1$ beneath the placer, starting from a position in which location i is beneath the placer, $1 \leq i \leq n-1$, and
$\Delta f(r, s)$ = time needed for a rack movement to bring feeder rack position r beneath the gripper, starting from a position in which feeder rack position s is beneath the gripper, $1 \leq r, s \leq K$.

We construct a graph G as follows. Let $F_i \subseteq \{1, \ldots, K\}$ be the index set corresponding to feeder rack positions which hold feeders from which component i may be retrieved. The graph has a source, a sink and n intermediate layers. Each layer i contains $|F_i|$ vertices, denoted by $v_j^i, j \in F_i, 1 \leq i \leq n$. Indeed, there is one vertex in layer i for each feeder from which component i may be retrieved. The interpretation of vertex v_j^i is the start of the grip (and place) activity which grips component i from feeder j. Each vertex in layer i has an arc going to each vertex in layer $i+1$, the source has an arc going to each vertex in layer 1 and each vertex in layer n has an arc going to the sink. There are no other arcs. See Figure 6.2 for a representation of G. The weight of an arc emanating from the source is 0, and the weight of an arc going to the sink is Δgp. The weight of an arc from v_r^i to v_s^{i+1}, $r \in F_i, s \in F_{i+1}, 1 \leq i \leq n-1$, equals:

$$\Delta gp + \max(\Delta c, \Delta t(i, i-1), \Delta f(r, s)).$$

Notice that when one interprets vertex v_r^i as the start of the grip (and place) activity which grips component i from feeder r, the weight of an arc defined above is equal to the time between two consecutive grips. (This follows from the description in Subsection 6.2.2).

Consider now any path in the graph G from the source to the sink. This path contains one vertex from each layer, reflecting a choice of feeders for the retrieval of the n components. Further, since the weight of an arc corresponds to the time between two consecutive grip activities, the length of the path equals the makespan of the machine. Also, it is easy to verify that each solution of the component retrieval

Figure 6.2: Graph G

problem corresponds to a unique path in G. Since one is interested in the shortest makespan, it follows that the component retrieval problem reduces to a shortest path problem on G, for which efficient algorithms are available (see for instance Ahuja, Magnanti and Orlin [1993]). (The recursive formulation given in Bard, Clayton and Feo [1994] would also lead immediately to a polynomial time algorithm for this version of the component retrieval problem.)

Finally, we restrict ourselves here to noticing that, even for a given component placement sequence and a feeder rack assignment, the computation of the makespan of a PCB is a nontrivial task. In fact, for other technologies the component retrieval problem may become substantially more difficult (see Crama, Flippo, van de Klundert and Spieksma[1995], Chapter 5).

6.3 The planning procedure

Let us now return to the general planning problem. Summarizing the discussion in the previous sections, we want to find:

(1). for each machine, a feeder rack assignment,

(2). for each board type, a component placement sequence on each machine, such that for each PCB of that type, the sequences form a partitioning of the components

required by the PCB,

(3). for each pair consisting of a machine and a board type, a component retrieval plan.

In this section we describe our planning procedure. This procedure is divided into two phases: Phase 1 determines a feeder rack assignment for each machine, and Phase 2 produces, for each pair consisting of a machine and a board type, a component placement sequence and a component retrieval plan, given the feeder rack assignment of Phase 1. Accordingly, this section contains two subsections each devoted to the description of a phase in the planning procedure. Let us remark here that the emphasis in the description of our planning procedure lies on Phase 1, since the feeder rack assignment problem for multiple board types has not been addressed in literature. For Phase 2 our description will be more superficial, due to the fact that we use standard (local search) techniques to obtain good component placement sequences.

Clearly, the planning procedure we present is hierarchical in nature: first, a feeder rack assignment is computed, next a component placement sequence and component retrieval plan are determined. Of course, the following question then arises: how do we evaluate the feeder rack assignment computed in Phase 1 without computing a placement sequence and retrieval plan, that is, without solving Phase 2? We deal with this issue by computing an *estimate* of the makespan of each board type on each machine given the feeder rack assignment and a corresponding partition of the components of each board type (see (2) in Section 6.1). These estimates are then used as an indication of the quality of the feeder rack assignments found.

Before describing Phase 1 in more detail, let us first start with some observations related to the placement machines described in the previous section.

Observation 1: A feeder is nothing but a tape containing components of a certain type. These tapes are expensive, and to keep inventory of these tapes low, it is desirable to restrict the number of tapes containing components of a certain type. In the plant considered, often a bound of 2 feeders for each component type was imposed, that is, no more than 2 feeders with the same component type can be used by all placement machines in the line. We follow this restriction in our planning procedure.

Observation 2: A basic characteristic, which has been observed by other authors as well (see Bard, Clayton and Feo [1994]), concerns the so-called *free movement*. To explain this, consider again Figure 6.2. Between two consecutive grip (or place) activities, the carousel must rotate, and this takes a certain amount of time. During this time, the feeder rack, as well as the table containing the PCB, can move without increasing the makespan. Since it is impossible to avoid carousel rotations, this phenomenon occurs between each pair of consecutive grip (or place) activities. These movements of the feeder rack and the table, taking place during a carousel rotation, are referred to as free movement. Of course, the significance of this effect depends on the magnitude of the free movements. However, this can be considerable. In our situation, free movements

of the feeder rack correspond to 1 position on the rack, that is, repositioning the feeder rack by 0 or 1 feeders is free. Concerning the table, free movement corresponds to approximately 2 cm on the table containing the PCB. Since the average PCB is approximately 20 × 30 cm, and the feeder rack contains mostly about 100 feeders, long table movements are less time consuming than long feeder rack movements. Hence we restrict the planning procedure to considering solutions in which all feeder rack movements are short, expecting that given this short feeder rack movements we can find a component placement sequence in which the table movements are not too long either. More specifically, we assume in Phase 1 that all components that are to be retrieved from feeder i must be retrieved interleavingly with the retrieval of components from feeders i and $i + 1$ (thereby utilizing the free movement as much as possible as far as the feeder rack is concerned). The solutions we obtain show that it is indeed possible to construct feeder rack assignments and component placement sequences such that subsequent components are mostly retrieved from consecutive feeders while the larger part of the table movements is free.

6.3.1 Constructing a feeder rack assignment: Phase 1

Let us now describe Phase 1. It consists of five steps.

Step 1: Determine which component types will have 2 feeders in the flowshop (see Observation 1).

Step 2: Decide, for each feeder, which locations it serves on each board type.

Step 3: Construct an arbitrary feeder rack assignment.

Step 4: Estimate the makespan for each board type on each machine, given the current feeder rack assignment.

Step 5: If some stopping criterion is satisfied, exit. Else, improve this feeder rack assignment using local search and go to Step 4.

Consider Step 1. In view of Observation 2 above, it is desirable for components that are within free gripping movement of each other, to be not too far apart from each other on the board, since otherwise the board movement (between consecutive placing operations) may take a long time. Thus, if there are more magazine rack positions than component types, one way to avoid long table movements is to assign two feeders to component types whose components are, on some boards, far apart. Of all strategies we have tested to utilize this idea the following simple strategy performed best. Compute, for each combination of board type and component type, a short Hamiltonian path through the corresponding locations using some (TSP) heuristic (we use farthest insertion). For each component type t, let l_t be the length of the longest edge occurring in some path corresponding to component type t. Next, we list the component types in order of decreasing l_t value and we assign two feeders to as many component

types as possible, starting at the top of the list and proceeding downward, until total capacity of the feeder rack is exhausted (or until each type has two feeders). In this way it is decided which component types have more than one feeder. Notice that we use individual board type characteristics to select those component types.

Step 2 resembles the component retrieval problem. The difference is that the position of the feeders in the rack is not fixed yet. To partition the locations corresponding to components of each type for which there are two feeders placed in the rack, we propose the following. First, we consider a board type which has lead us to assign two feeders to a component type t: consider board type p on which the Hamiltonian path as computed in Step 1 contains an edge of length l_t. Removing this longest edge partitions the locations corresponding to components of type t on board type p into two subsets, say $L_1^{t,p}$ and $L_2^{t,p}$.

We have to decide next how to partition the components of this type on boards other than boards of type p. We take the following approach. For each location corresponding to a component of type t not on board type p, we determine the minimal distance to some location in $L_1^{t,p}$ and, similarly, we determine the minimal distance to some location in $L_2^{t,p}$. Next, we assign this location to the subset whose corresponding minimal distance is minimal. (Notice that we assume here that a distance is known between two locations which do not occur on the same board type. These distances are computed as if the components where on a same board, as e.g. the composite board mentioned earlier. This composite board can be constructed unambigously since all component locations are expressed in terms of coordinates, and the position of the boards on the table are given.) This approach has outperformed several alternative approaches in our computational experiments. Its succes should be contributed to the fact that in this way, for each board type, the set of locations to be served by a feeder lies in the same part of the board.

In this way, we obtain two sets of locations for each component type that has two feeders in the flowshop. More precisely, we refer to this partitioning of locations corresponding to components of a certain type as a *clustering*, and we refer, informally, to all components that are to be retrieved from the same feeder as a *cluster*. In Figure 6.3 we give a more formal description of the algorithm described in steps 1 and 2.

In Step 3 we simply assign the feeders arbitrarily to some position in the racks of the machines such that a feasible feeder rack assignment is obtained.

Consider Step 4. Recall from Observation 2 that, given a feeder rack assignment, we intend to construct component placement sequences with the property that the rack moves at most one position between any two consecutive grip activities. In addition, when estimating the makespan of a board given a feeder rack assignment, we assume that one pass of the feeder rack through all feeders gives a good approximation of an optimal way of moving the feeder rack. More explicitly, we estimate the makespan by

L^t: set of locations corresponding to components of type t.
$L^{t,p}$: set of locations corresponding to components of type t on a board of type p.
$P^{t,p}$: sequence of locations corresponding to components of type t on a board of type p induced by a Hamiltonian path.
$d_{ij}^{t,p}$: distance between location i and location j corresponding to components of type t on a board of type p.
$dis(i, L)$: minimum distance over the locations in the set L between location i and a location from L.
$\#t$: number of component types.
$\#p$: number of board types.
cap: number of feeder rack positions in the line.

(1). for $t := 1$ to $\#t$ do
for $p := 1$ to $\#p$ do
construct a Hamiltonian path $P^{t,p}$ using distance matrix $d^{t,p}$;
Let the value of the longest edge in $P^{t,p}$ be denoted by $z_{t,p}$;

(2). for $t := 1$ to $\#t$ do

(a) $l_t :=\max_p z_{t,p}$;

(b) $p_t :=\arg\max_p z_{t,p}$;

(3). sort(l_t) in decreasing order; $q := \#t$; $t := 1$;

(4). while $q < \min(\text{cap}, 2 \times \#t)$ do
comment: 2 feeders for type t
$t := t + 1$; $q := q + 1$;

(5). $t^* := t - 1$; *comment: t^* corresponds to the number of feeder duplications*

(6). for $t := 1$ to t^* do

(a) partition L^{t,p_t} into L_1^{t,p_t} and L_2^{t,p_t} by removing the longest edge from P^{t,p_t};

(b) $C_{2t-1} := L_1^{t,p_t}$; $C_{2t} := L_2^{t,p_t}$;

(c) for each $i \in \cup_{p \neq p_t} L^{t,p}$ do
if $dis(i, L_1^{t,p_t}) < dis(i, L_2^{t,p_t})$ then $C_{2t-1} := C_{2t-1} \cup \{i\}$ else $C_{2t} := C_{2t} \cup \{i\}$;

(7). for $t := t^* + 1$ to $\#t$ do $C_{t^*+t} := L^t$;

Figure 6.3: Algorithm for steps 1 and 2

assuming that the feeder rack movement in an optimal solution follows approximately the following pattern: the rack starts at the feeder in the left most position, to be called feeder 1. Then, all components are gripped (and placed) of the cluster assigned to feeder 1, interleaving them by gripping (and placing) components of feeder 2, ending with a grip of a component from feeder 2. This is followed by gripping (and placing) the remainder of the components that are to be gripped from feeder 2, interleaved with the gripping of components of feeder 3 et cetera.

This leads to a solution in which all feeder rack movements are free (if at least one component is retrieved from each feeder). Therefore, the makespan depends on the length of the table movements only, i.e. the length of a Hamiltonian path through the locations, that respects the pattern of the feeder rack movements described above. For our purposes it is desirable to be able to compute quickly an estimate of the length of such a Hamiltonian path. Therefore we propose the following method, that does not use the exact feeder rack assignment, but only knowledge of which feeders are assigned to adjacent positions in the feeder rack. Notice that for every pair of feeders there is a set of locations on a board where the components from these feeders will be placed. We compute, for every pair of feeders, a value which equals the length of a Hamiltonian path through these locations, thereby utilizing the fact that they may be gripped interleavingly. Notice that these quantities can be computed independently of a feeder rack assignment, reducing the amount of computations required to evaluate individual feeder rack assignments.

Of course, it may well be the case that, given a feeder rack assignment and some board type, there is a set of adjacent feeders from which no component is taken at all. In that case, the feeder rack movement is not free, and the makespan depends on the duration of this feeder rack movement. We next present an algoritm to estimate the makespan of every board type in the family for a given feeder rack assignment on some machine.

We compute, for each board type p in the family, two so-called cluster distance matrices D^p and E^p. D^p_{ij} represents the length of a (short) Hamiltonian path through all locations of clusters C_i and C_j that are to be placed on p. E^p_{ij} is the maximum of two numbers, E^{p1}_{ij} and E^{p2}_{ij}, where E^{p1}_{ij} is the minimum distance over all distances between locations of C_i and C_j on board type p and E^{p2}_{ij} corresponds to the distance between the feeders positioned in the rack which correspond to clusters C_i and C_j. Notice that these computations must be performed efficiently, since for the problem instances described in Section 4, the computation of the matrix D requires constructing several hundreds of thousands Hamilton paths.

Now, to estimate the makespan of a board on a machine, we sum, for every pair of consecutive feeders from which components are gripped, 1/2 of the length of the Hamiltonian path through the locations of both corresponding clusters (this value is stored in D^p), while taking into account those feeders from which no components are gripped (using values in E^p). A more formal description of the algorithm estimating the makespan of board type p on a machine with a given feeder rack assignment is as given in Figure 6.4.

π: a permutation of the feeders, i.e the order in which the feeders are placed on the rack (say from left to right),
C_i^p: the set of components from cluster C_i (the cluster corresponding to the feeder positioned in $\pi(i)$) that have to be placed on board type p.

(1). $i := 1, span := 0$;

(2). while $|C^p_{\pi(i)}| = 0$ do $i := i + 1$;

(3). $span := span + 1/2 \times D^p_{\pi(i),\pi(i)}$;
$i := i + 1$;

(4). while $|C^p_{\pi(i)}| > 0$ do

(a) $span := span + 1/2 \times D^p_{\pi(i-1),\pi(i)}$;

(b) $i := i + 1$;

(c) if i = total number of feeders then set $k = i$ and goto step 7;

$k := i - 1$;
$span := span + 1/2 \times D^p_{\pi(k),\pi(k)}$;

(5). while $|C^p_{\pi(i)}| = 0$ do

(a) $i := i + 1$;

(b) if i = total number of feeders then goto 7;

(6). $span := span + E^p_{\pi(k),\pi(i)}$;
goto 3;

(7). $span := span + 1/2 \times D^p_{\pi(k),\pi(k)}$;

Figure 6.4: Makespan estimation algorithm

Using the algorithm in Figure 6.4, we get an estimate of the makespan of every board type for a given feeder rack assignment for each machine in the flowshop, without having to specify a component placement sequence. Based on these estimated makespans, we can also get an estimate of the objective function for a given feeder rack assignment for each machine. Of course, the accuracy of such an estimate depends on the quality of the Hamiltonian paths constructed, and on the component placement sequences that will eventually be found. For a further discussion of these estimates, see Section 6.4.

Step 5 of our solution procedure optimizes the feeder rack assignments, using the estimated objective function value found in Step 4. In fact, it is in this step that our estimates are used extensively. Throughout this step objective function evaluations are based on the estimates of the makespans, instead of the actual makespans. We try to optimize the feeder rack assignment by using two heuristics alternately.

One heuristic tries to exchange between two machines a pair of clusters, together with the corresponding feeders, to better balance the workload. To determine which pair of machines are candidates for exchanging clusters we do the following. For each pair of machines we sum over all board types the absolute difference of the respective processing times. Next, we select that pair of machines for which this sum is maximal, and attempt to improve our current feeder rack assignment by exchanging feeders (and corresponding clusters) between these machines. Now what do we mean by improve? Obviously, what we want to improve is the objective function as described in Subsection 6.2.1. However we have chosen another surrogate objective function to speed up the heuristic. Let M_i and M_j be the machines between which feeders are being exchanged, then the surrogate objective function takes on the value which the objective function would have when machines M_i and M_j are the only machines in the flowshop.

The other heuristic reoptimizes the feeder rack assignment for a single machine. More precisely, for a given machine, it attempts to minimize the sum of the makespans of the boards on that machine, by optimizing the feeder sequence. This sequencing problem is solved using an insertion heuristic in which the makespan estimation algorithm given in Figure 6.4 is used to evaluate the insertions.

Together these two heuristics deliver better solutions faster than other approaches we have tested. Since the heuristics work with different objective functions, which are in turn different from the original objective function, the process of calling both heuristics in turns does not neccesarily converge. Therefore we have the following stopping criterion. We specify some upperbound (say 5 secs.), and the algorithm stops if after some iteration, the maximum over all pairs of machines of the sum of the absolute differences of their makespans does not exceed this upperbound. This strategy implies that when the algorithm stops, there may still be some room for improvement of the balance. Achieving these last tenths of procents of improvement is time consuming, and becomes less attractive as one notices that this phase of the algorithm uses estimations rather than the actual makespans.

6.3.2 Constructing a component placement sequence: Phase 2

Phase 2 of the solution strategy is to determine, for each machine - board type combination a component placement sequence and an associated component retrieval plan. We employ constructive methods as well as local search algorithms to find a component placement sequence. Given this sequence, an optimal component retrieval plan can be found using a shortest path algorithm as described in Subsection 6.2.3.

For a given board type and machine, we construct an initial component placement sequence as follows. First assume that each location is served from the feeder associated to the cluster containing that location. For the first two feeders, determine two locations, c_1 (from the left most feeder, feeder 1) and c_2 (from the feeder adjacent to it, feeder 2), that are nearest to each other. Then sequence before c_1, all locations that are to be served from the left most feeder as well, using some insertion heuristic. Next determine the two locations c_2' from feeder 2, and c_3' from feeder 3, that are nearest to each other, and sequence all remaining locations that must be served from feeder 2 between c_2 and c_2' et cetera. Given this sequence we solve the component retrieval problem, which provides a first solution. Recall that the component retrieval problem is essentially a shortest path problem.

Next, we try to improve the placement sequence by TSP-like local search techniques, using 2opt and a restricted version of 3opt. This local search process may be very time consuming since each local search step requires solving the component retrieval problem. Therefore we have sought ways to speed up this local search phase. An easy way of doing so, without substantially influencing the effectiveness of the 2opt heuristic, is to keep the component retrieval plan fixed. In this way, the time consuming resolving of the component retrieval problem in each iteration of the 2opt heuristic can be skipped. (It should be noted however, that each iteration still requires performing some non-negligible computations due to the fact that changes in the component placement sequence will change for some components, the component that is gripped while it is being placed.) On the other hand, especially for the restricted 3opt heuristic (that essentially takes out one component of the placement sequence and then tries to reinsert it) resolving the component retrieval problem may be well worth the additional effort. We have tried to reduce the running time of the local search heuristics by implementing several ideas that keeps them from considering or evaluating (by solving the component retrieval problem) moves that will not result in an improvement.

We aimed to keep the running time of the entire algorithm (phases 1 and 2) within certain limits. As a consequence the algorithm cannot spend too much time optimizing the component placement sequence of each board, even though the local search heuristics usually improve the solutions significantly. The methods described above to speed up these heuristics ensure that the benefits of these heuristics are realized.

Notice that the feeder rack movements resulting from the final component placement sequences may not always utilize free movements as the intended feeder rack movements in Phase 1 of the solution approach did. Further, notice that the clustering found in

steps 1 and 2 of Phase 1 can be changed by the component retrieval plan.

6.4 Computational results.

In this section we test our planning procedure on two datasets. Dataset 1 corresponds to a family consisting of 9 board types assembled by a line of 3 CP-IV machines. Dataset 2 corresponds to a family consisting of 7 board types assembled by a line of 2 machines, a CP-IV and a CP-III machine. These datasets are real-life data made available to us by Philips. The planning procedure we described above is programmed in Turbo Pascal and run on a personal computer with a 486 33MHz processor. The results of our procedure are described in Tables 6.1 to 6.6.

board type	Machine 1		Machine 2		Machine 3	
	NoC	time	NoC	time	NoC	time
1	50	13.8	52	13.5	49	13.5
2	49	13.8	52	13.5	50	13.8
3	86	21.1	86	21.2	84	21.2
4	66	17.4	72	18.4	70	18.5
5	25	7.9	28	8.0	25	7.0
6	25	7.9	28	8.2	26	7.5
7	42	11.3	43	12.2	47	13.4
8	58	15.7	61	16.1	60	16.9
9	304	69.4	293	69.3	332	71.0

	time
Total makespan	184.8
Current solution	244.1
Lower bound	154.8

Table 6.1: Final results Dataset 1

To explain Table 6.1, consider a column corresponding with a machine. An entry in this column has two numbers: "NoC" is the number of components of the specific board type placed by that machine, and "time" equals the number of seconds it takes to place these components by this machine. The total makespan is computed by summing over the board types, the makespan of these boards on their bottleneck machines. This total makespan is compared with the makespan of the solutions obtained by Philips (referred to as current solution) and with a lower bound. This lower bound is computed as follows. Recall that Δc represents the time a single carousel shift takes; further, let *totcomp* be the total number of components to be placed to produce a single board of each type in the family, and let m equal the number of machines in the line. Then $\frac{\Delta c \cdot totcomp}{m}$ is a valid lower bound for the makespan. For dataset 1, we are able to

improve the current solution by almost a minute, closing 66% of the gap between the lower bound and the current solution.

board type	Machine 1		Machine 2		Machine 3	
	actual	estimated	actual	estimated	actual	estimated
1	13.8	13.6	13.5	13.1	13.5	13.5
2	13.8	13.5	13.5	13.1	13.8	13.6
3	21.1	22.2	21.2	21.3	21.2	22.2
4	17.4	18.8	18.4	18.5	18.5	19.2
5	7.9	8.2	8.0	8.4	7.0	7.4
6	7.9	8.1	8.2	8.4	7.5	7.6
7	11.3	12.3	12.2	12.8	13.4	13.3
8	15.7	16.4	16.1	17.2	16.9	17.0
9	69.4	71.2	69.3	72.2	71.0	71.6

Table 6.2: Estimated Results Dataset 1

In Table 6.2 the estimates of the makespans as computed in Phase 1 are presented. We conclude that these estimates are accurate enough (usually within a few percent) to give a realistic impression of the actual makespans delivered by Phase 2. This might for example be useful to evaluate alternative solutions to the partitioning of board types into families (problem (1) in Section 6.1).

Table 6.3 to 6.6 arise as follows. As mentioned in Subsection 6.2.2, the difference between a CP-III and a CP-IV is its speed. To model this difference, we assume that speed (CP-IV) = speedfactor * speed (CP-III). Since this speedfactor (spf) is a simplification of reality it is hard to estimate exactly. The speedfactor is believed to be approximately 15%, but Table 3 also shows the outcomes for other values. Although it is hard to make an exact statement regarding to the outcomes of our algorithm based on Table 3, it shows that our planning procedure is quite robust. The makespan grows more or less proportionally to the speedfactor. Moreover, each of the four solutions exceed the lower bound by not more than 7%, while the current solution exceeds the lower bound by more than 20%.

Concerning the topic of running times, we restrict ourselves to the following general remarks. The running time of Phase 1 of the planning procedure (that is, constructing a feeder rack assignment) varies. Due to the fact that it is much harder to balance three machines than two, step 5 takes much more time for dataset 1 than for dataset 2. The exact running time depends on the dataset and the stopping criterion, but Phase 1 takes approximately 15 minutes on a 33Mhz 486. For both instances, Phase 2 took roughly about 10 to 15 minutes. In practice the amount of computation time needed does not seem to be severely restricted, since feeder rack assignments are not changed frequently. In fact, the running times given above are in the same order of magnitude as the running times of the existing software.

spf = 1.15 board type	Machine 1		Machine 2	
	NoC	time	NoC	time
1	107	28.7	129	29.8
2	138	37.1	164	37.1
3	137	37.2	162	37.0
4	306	80.7	378	82.6
5	118	31.3	139	31.6
6	149	39.2	174	39.5
7	150	39.5	172	38.8

	time
Total makespan	297.3
Current solution	361.8
Lower bound	284.2

Table 6.3: Dataset 2, speedfactor 1.15

spf = 1.20 board type	Machine 1		Machine 2	
	NoC	time	NoC	time
1	104	29.4	132	31.0
2	134	37.3	168	39.1
3	131	37.2	168	38.9
4	303	82.2	381	84.6
5	112	31.7	145	33.7
6	144	40.4	179	40.9
7	144	39.4	178	40.8

	time
Total makespan	309.0
Current solution	361.8
Lower bound	289.8

Table 6.4: Dataset 2, speedfactor 1.20

spf = 1.25 board type	Machine 1		Machine 2	
	NoC	time	NoC	time
1	102	30.2	134	32.1
2	132	38.2	170	39.4
3	131	38.2	168	39.1
4	299	84.1	385	86.4
5	110	33.0	147	34.2
6	142	41.5	181	41.5
7	140	41.1	182	41.3

	time
Total makespan	314.0
Current solution	361.8
Lower bound	295.2

Table 6.5: Dataset 2, speedfactor 1.25

spf = 1.30 board type	Machine 1		Machine 2	
	NoC	time	NoC	time
1	99	31.3	137	31.1
2	128	39.5	174	39.1
3	126	39.2	173	38.7
4	285	84.6	399	87.7
5	105	31.1	152	34.3
6	138	42.7	185	41.4
7	137	41.0	185	41.4

	time
Total makespan	315.9
Current solution	361.8
Lower bound	300.3

Table 6.6: Dataset 2, speedfactor 1.30

Summarizing, the tables show that the solution procedure we present yields, at least in terms of the makespan, significant better results than the existing software. In our view, this difference is caused by the fact that we solve the feeder rack assignment problem using individual board characteristics contrary to existing software which uses a composite board type. Another (small) advantage of the solutions found by our approach is that the movements of the feeder rack tend to be relatively small. This causes less wear for the rack.

6.5 Conclusions

This chapter deals with the assembly of a family of board types by a single line of placement machines. By decomposing the planning problem, a number of subproblems arise. An important subproblem is then to construct a feeder rack assignment for each of the machines which allows us to construct good placement sequences for each of the board types in the family. Here, we explicitly address this problem and we propose a heuristic based on the individual board characteristics. This heuristic is incorporated into a solution procedure which delivers a solution for the general planning problem. The computational results show that this approach works well.

Part III

Tool management

Chapter 7

The approximability of tool management problems

7.1 Introduction

Regardless of the precise definition of flexibility in the term flexible manufacturing systems, the ability of machines to perform various operations on various products or parts, is a most vital component of this flexibility. This flexibility of the machines is achieved by equipping them with a tool magazine, which enables the machines to hold a set of tools from which, depending on the operation the machine has to perform, it uses one tool or another. The resulting flexibility may be advantageous from a strategic or even tactical viewpoint, it comes at a price. The complexity of the operational planning and scheduling of the machines increases considerably, even when considering the machines in isolation. Apart from the part sequencing decisions, that normally constitute a solution to a single machine scheduling problem, one has to specify tool handling decisions. For this reason, problems concerning the scheduling of a single flexible machine, which are so fundamental to the scheduling of flexible manufacturing systems, differ essentially from classical single machine scheduling problems. It should therefore not come as a surprise that since the introduction of these machines, they have received considerable attention in the literature.

A good deal of the literature concerning tool management problems in single machine scheduling problems takes a practical position. The authors extract a problem from a more or less real life situation and propose an approximate solution strategy. Other authors, by contrast, are interested in the mathematical models underlying one or several of these problems. In this chapter we are also interested in mathematical properties of single flexible machine scheduling problems. More specifically, we will be interested in the worst case ratios of approximation algorithms for single flexible machine scheduling problems. This means that we borrow from both theoretical as well as applied papers. Many of the algorithms as they are proposed in the literature have appeared in applied papers. On the other hand, a proper classification of the complexity of the models and their approximability requires a more theoretical background.

Few attempts have been made to date to classify the problems discussed in this chapter with respect to their approximability. Rajagopalan [1985] establishes that a simple 'First Fit Decreasing' heuristic 'can do almost arbitrarily bad' for certain batching problems. Kortsarz and Peleg [1993] consider a special case of the batch selection problem that may be interpreted as the problem of finding a densest induced subgraph. They present an approximation algorithm for finding a dense subgraph of a graph $G(V, E)$ of cardinality at most C, whose worst case ratio is $O(|V|^{\frac{7}{18}})$. Goldschmidt et al. [1992] also propose several approximation algorithms for special cases of both the batch selection and the related job grouping problem, with constant worst case ratios or worst case ratios that are linear in the tool magazine capacity C. Goldschmidt et al. [1993] suggest a dynamic programming formulation for the batch selection problem and discuss conditions under which it can be implemented to run in polynomial time.

In the next section, we first discuss several of the basic single flexible machine scheduling problems and we also briefly discuss some mathematical models, their complexity, and their relationships with other combinatorial problems. Section 3 investigates the worst case behavior of several algorithms as they are proposed in the literature to date. This investigation will lead us to the conclusion that all of these algorithms have very poor worst case behavior. However, as yet it is not known whether there (can) exist polynomial approximation algorithms with a better worst case behavior. We give negative results addressing this question. In section 4 we conclude by discussing directions for further research.

7.2 Models and complexity

In this section we briefly discuss several models as they arise naturally in the context of flexible machine scheduling, and their complexity. Our main purpose is to facilitate the analysis in the subsequent sections. For a more detailed treatment of the models we refer to Crama [1995].

To start with, let us take a look at the physical characteristics of flexible machine scheduling. First of all, there is a *machine* on which a set of *jobs* or *parts* have to be processed. (We use jobs and parts interchangeably). Processing means that the machine performs one or several operations on these jobs, and the execution of each of these operations requires one or more *tools*. The machine can store tools in its *tool magazine*. In this chapter, we assume the magazine contains C slots, and that each tool requires exactly one slot, although more general models are possible of course (Crama [1995]). Using tools from the tool magazine requires little set up time, and thus, as long as a sequence of jobs that have to be processed only requires tools that are present on the machine, in the tool magazine, set up times are (negligibly) small. However, if the number of tools required by a sequence of jobs exceeds the *tool magazine capacity*, it is unavoidable that some tool is removed from the magazine to be replaced by another

tool. We refer to such an event as a *switch*. Switches usually take nonnegligible time, and thus whenever this happens set up times are 'large'.

Under many reasonable objective functions of such scheduling problems, e.g. makespan minimization, we have to minimize the sum of the set up times. The total set up time is usually computed in one of the two following ways. If tools cannot be switched simultaneously, total set up time depends linearly on the number of tool switches. On the other hand, supposing that tool switches may be performed (completely) simultaneously, total set up time depends linearly on the number of switching instants. Regardless of the definition of total set up time, let us define a *loading strategy* as a specification of the contents of the tool magazine over time. We now identify the following four basic scheduling problems, whose names are taken from Crama [1995]:

(1). **Tool Switching :** The problem of finding an input sequence for the parts and a loading strategy for the tool magazine with minimum total set up time, in case total set up time depends linearly on the number of switches.

(2). **Loading Problem :** The problem of finding for a given part input sequence, a loading strategy with minimum total set up time, in case total set up time depends linearly on the number of switches.

(3). **Job Grouping :** The problem of finding an input sequence for the parts and a loading strategy for the tool magazine with minimum total set up time, in case total set up time depends linearly on the number of switching instants.

(4). **Batch Selection :** The problem of finding the largest group of jobs that can be processed without tool switches.

We have enumerated here the optimization versions of the four problems, but we refer to the decision versions by the same name.

The Loading Problem is a special case of the Tool Switching Problem that is interesting not only from a computational viewpoint ; both the Tool Switching problem and the Loading Problem also arise in the context of (mainframe) computer memory management [Blazewicz & Finke 1994].

Notice that the only difference between the Tool Switching Problem and the Job Grouping problem is the underlying cost structure. However, the Job Grouping Problem is in a way less sensitive to the exact part input sequence. Given two consecutive switching instants, the order in which the jobs are processed between these instants does not matter. Let us therefore call a set of jobs that can be processed without tool switches, a *batch*. Then, the Job Grouping Problem boils down to finding a partitioning of the jobs in a minimum number of *batches*. These observations motivate our interest for the fourth problem, the Batch Selection Problem. Sometimes batches are referred to as groups, which explains the name Job Grouping.

Without going into any further detail of the mathematical models that exist for the four problems, let us discuss their complexity status first. Crama, Kolen, Oerlemans

& Spieksma [1994], show that the Tool Switching Problem is NP-hard, even for fixed $C \geq 2$. In computer science it has been long known that the Loading Problem can be solved in polynomial time. This problem is also investigated by Tang & Denardo [1988], Crama, Kolen, Oerlemans & Spieksma [1990], and Privault & Finke [1993] who provide a network flow formulation.

The Job Grouping Problem has also been shown to be NP-hard by several authors. Crama & Oerlemans [1994] show that the problem is NP-hard, even for fixed $C \geq 3$, and that it is NP-hard to decide whether there exists a partitioning of size two. By showing that the well known Set Covering Problem may be viewed as a special case of the Job Grouping Problem in which all maximal batches are known, they also establish that the problem remains NP-hard in such a case. In general, however, the problem is in a way even harder, since the Batch Selection Problem is known to be NP-hard even when each part requires two tools [Gallo, Hammer & Simeone 1980].

In this chapter we will be primarily interested in the Job Grouping Problem and in the Batch Selection Problem. We discuss the relationship between the Job Grouping and the Batch Selection problem at length in subsequent sections. The Loading Problem is polynomially solvable, making the study of approximation algorithms for this problem less interesting. The approximability of Tool Switching is discussed in relation to the approximability of Job Grouping.

For the moment we restrict the analysis to Job Grouping and Batch Selection and we first make clear that both problems require the same data. An instance consists of a tool magazine capacity C, a set J of jobs $p_1, \ldots, p_n$ and a set T of tools $t_1, \ldots, t_m$. Further, for each job $p_i, i = 1, \ldots, n$ we specify which tools it requires. To this purpose, we introduce a $(0,1)$ matrix A, whose m rows correspond to tools and whose n columns correspond to the jobs. Naturally, we let $a_{ij} = 1$ if tool t_i is required by job p_j, and zero otherwise. Thus, an instance of (an optimization version of) Job Grouping or Batch Selection is completely specified by a $(0,1)$ matrix A and a positive integer C, the tool magazine capacity. In the remainder, we assume that for each pair of jobs, the sets of tools required for each of the jobs do not contain one another as a subset.

As such, the Batch Selection problem is to find a maximum cardinality subset J of the columns, such that $|\{i | \sum_{j \in J} a_{ij} > 0\}| \leq C$. The matrix A, which we shall refer to as the *tool-job matrix*, may also be viewed as the node incidence matrix of a hypergraph $H(V, E)$. Each row i corresponds to a vertex $v_i \in V$, and each column j to a hyperedge $e_j \in E$. Indeed, $a_{ij} = 1$ indicates that edge e_j contains vertex v_i. In this setting the Batch Selection problem is to find a densest subset of the vertices of cardinality at most C, i.e. a subset of the vertices of cardinality C, whose induced subgraph contains the largest number of hyperedges. Now, in the decision version of Clique, one may have to find a subset V' of the vertex set, with $|V'| = C$, such that the subgraph induced by V' contains $\frac{1}{2}|V'|(|V'|-1)$ edges. This establishes that Batch Selection is already NP-Complete when each job requires two tools [Gallo, Hammer & Simeone 1988].

Of course, Job Grouping may also be interpreted in terms of $(0,1)$ matrices and hypergraphs. One has to find a minimum cardinality set of subsets $S_1, \ldots, S_K$ of the vertices, such that the subhypergraphs $H^i(S_i, E_{S_i})$ induced by these subsets S_i form a covering of H, i.e. $\cup_i E_{S_i} = E$. As observed before, this problem is already NP-hard when all maximal subhypergraphs are known. In such a case we obtain an ordinary covering problem as follows. We introduce a job-group matrix B, in which the rows correspond to jobs and the columns correspond to groups. Indeed $b_{jk} = 1$ if job p_j is in group g_k, and zero otherwise. Letting the matrix B be the constraint matrix, the Job Grouping problem is then turned into a Set Covering Problem.

We informally conclude that the Job Grouping Problem does not appear to be easier than Set Covering, and that the Batch Selection Problem is closely related to Clique. Both Set Covering and Clique are notoriously hard from a viewpoint of approximation. We discuss their exact status in Section 3. For the moment however, we should have modest expectations with respect to the worst case ratios of polynomial approximation algorithms for Job Grouping and Batch Selection as they are given in the next section.

7.3 Approximation algorithms and worst case ratios

In this section we overview approximation algorithms for Job Grouping and Batch Selection, as they have been proposed in the literature. We also study their worst case behavior.

From the literature we have extracted the following list of approximation algorithms for the Batch Selection problem. Each algorithm is characterized by a selection rule, which specifies how to select jobs or tools. The selection rule is to be repeatedly applied as long as some stopping criterion is not satisfied, e.g. the total number of tools (required by the selected jobs) does not exceed the tool magazine capacity.

(1). MIMU rule (Tang and Denardo [1988b]) : Select the job that has the largest number of tools in common with the jobs already in the batch. In case of a tie, select the job which requires the smallest number of additional tools : Maximal Intersection Minimal Union.

(2). MI (Maximal Intersection) rule, break ties arbitrarily.

(3). MU (Minimal Union) rule, break ties arbitrarily.

(4). Whitney & Gaul rule (Whitney & Gaul [1985]) : Let $t(Y)$ be the number of tools required by the jobs in the set of jobs Y. Let B be the set of already selected jobs. Select the job p that maximizes $(t(B \cup \{p\}) + 1)/(t(\{p\}) + 1)$.

(5). Rajagopalan rule (Rajagopalan [1985]) : Define the weight of each tool to be the number of jobs that require it among the jobs not yet assigned to the batch. Select the job for which the sum of the weights of the tools that are to be added when this job is selected is maximum.

(6). Modified Rajagopalan rule (Crama & Oerlemans [1992]) : Define the weight of a tool to be the number of jobs already selected that require this tool. Select the job for which the sum of the weights of the tools needed by this job is maximum.

(7). Chaillou, Hansen & Mahieu [1989] rule : Create an initial batch consisting of all jobs by selecting all tools. Then, iterate deleting tools from the set of selected tools until the number of selected tools equals the magazine capacity. In each iteration delete the tool which causes the smallest number of jobs to be eliminated from the batch.

(8). Marginal gain rule (Dietrich, Lee & Lee [1991]) : Define the weight of a job to be the number of jobs that can be added without tool addition when this job is selected. Select the job with maximum weight.

Rajagopalan [1985] also proposed a rule of the Maximal Union type, and showed that it can perform arbitrary bad on certain instances. This was the motivation to introduce rule 6 described above.

Every approximation algorithm A_{BS} for the Batch Selection problem can be viewed as an approximation algorithm for Job Grouping A_{JG} : in order to solve the Job Grouping problem we select groups in the following manner. We apply A_{BS} to find a first group. Then we eliminate the jobs in this group formed by A_{BS} from the instance and apply A_{BS} again. We repeat this procedure until there are no jobs left. The sequence of groups that is output by A_{BS} form a solution of the Job Grouping problem on the same data. As a matter of fact, all heuristics for the Job Grouping problem known to the authors are of this type.

In the remainder of this section, we analyse the worst case ratio of the rules given above. In the worst case instances we present, the optimal solution of the Job Grouping problem consists of a set of batches that all are optimal solutions to the Batch Selection problem. Furthermore, repeatedly applying a selection rule, results in selecting a set of batches that all contain the same number of jobs. Hence the worst case performance of a rule on the Job Grouping problem is the reciprocal of the worst case ratio of this rule on the Batch Selection problem. Therefore we primarily consider the behavior of the rules on the Batch Selection problem.

Before we analyse the worst case behavior of the proposed heuristics, let us present an upperbound on the worst case ratio of any heuristic (for the Batch Selection problem). For a magazine capacity C, the maximum number of jobs in a group can be seen to be $\begin{pmatrix} C \\ C/2 \end{pmatrix} = \Omega(\frac{2^C}{\sqrt{C}})$, whereas any heuristic finds a group of at least a single

$$\left(\begin{array}{cccccc|cccccc|cccccc}
1&1&1&0&0&0&1&1&1&0&0&0&1&1&1&0&0&0\\
1&0&0&1&1&0&1&0&0&1&1&0&1&0&0&1&1&0\\
0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\
0&0&1&0&1&1&0&0&1&0&1&1&0&0&1&0&1&1\\
\hline
1&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&1&1&1&1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1
\end{array}\right)$$

Figure 7.1: Tool job matrix of a worst case instance for heuristics 1,2,3,6 ($k = 4$)

job. Hence, $\binom{C}{C/2}$ is an upperbound on the worst case ratio of any heuristic. We show here that rules 1 to 6 have worst case ratios in this order of magnitude, and may therefore perform arbitrarily bad :

Theorem 7.1 The worst case ratio of heuristics 1,2,3, and 6 is at least $\Omega(\frac{2^C}{C^{3/2}})$ and $\Omega(n/\log^2 n)$.

Proof. Let k be some even integer. We create instances in which there are k top tools and each job requires $k/2$ of them. We first consider an instance in which there are $k/2+1$ bottom tools, of which each job requires only 1. The tool magazine capacity $C = k+1$. We have a set of $k/2+1$ jobs for each possible choice of $k/2$ top tools, one job in the set for each possible bottom tool. Thus we have $\binom{k}{k/2} \times (k/2+1)$ jobs. (see Figure 7.1.) Obviously, for the Batch Selection problem, an optimal batch is the set of jobs requiring the same bottom tools. Moreover, the optimal solution for the Job Grouping problem is to form $k/2+1$ groups (each of them an optimal batch), one for each bottom tool.

It is not hard to see that heuristics 1,2,3 and 6 start with an arbitrary job and may subsequently select the job requiring the same top tools but another bottom tool. In this way they obtain batches of size $k/2+1$, where the optimal batches consist of $\binom{k}{k/2}$ jobs. The ratio between the number of jobs in an optimal batch and the number of jobs in a batch found by either of the heuristics is $\Omega(\frac{2^C}{C^{\frac{3}{2}}})$. Notice that in this instance selecting jobs randomly could not have led to a worse solution, be it for the Batch Selection problem or for the Job Grouping problem. ∎

Theorem 7.2 Heuristics 4,5 have worst case ratio of $O(\frac{2^C}{\sqrt{C}})$ and $O(n)$.

$$\left(\begin{array}{cccccc|cccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1
\end{array}\right)$$

Figure 7.2: Tool job matrix of a worst case instance for heuristics 4 and 5, ($k = 4$)

Proof. Heuristics 4 and 5 realise the ratio claimed in the theorem on the following instance. Let k again be some even integer. Again, there are two sets of tools, top tools and bottom tools. Each job requires either some of the top tools or some of the bottom tools. Hence we can also speak of toptool jobs and bottomtool jobs. There are k top tools and k bottom tools, and each job requirs $k/2$ tools, i.e. $k/2$ top tools or $k/2$ bottom tools. The tool magazine capacity $C = k$. We have a job for each possible choice of $k/2$ top tools out of k top tools, and a job for each possible choice of $k/2$ bottom tools. Thus we have $\left(\begin{array}{c} k \\ k/2 \end{array}\right) \times 2$ jobs. (see Figure 7.2.) Obviously, both the set of all toptool jobs and the set of all bottomtools jobs are optimal solutions to the Batch Selection problem. Moreover, the optimal solution for the Job Grouping problem is to form 2 groups (each of them an optimal batch).

It is left to the reader to check that rules 4 and 5 may pick batches of size 2, consisting of a toptool job and a bottomtool job, whereas the optimal batches consist of $\left(\begin{array}{c} k \\ k/2 \end{array}\right)$ jobs, yielding a worst case ratio of $O(\frac{2^C}{\sqrt{C}})$ and $O(n)$. ■

Notice that the bounds of Theorem 7.2 realize the aforementioned upperbound on the worst case ratio of any algorithm. In case all jobs require the same number of tools, rule 4 boils down to the Maximum Union rule. Rajagopalan already analyzed this Maximum Union rule and showed that it can do 'arbitrarily bad'.

Rule 7 does not solve the instances proposed in the proofs of Theorem 7.1 and Theorem 7.2 to optimality, but it does not perform as poorly as rules 1-6.

Theorem 7.3 Heuristic 7 has worst case ratio of at least $\Omega(\left(\begin{array}{c} \sqrt{C} \\ \frac{\sqrt{C}}{2} \end{array}\right)/\sqrt{C})$ and $\Omega(n/\log n)$.

Proof. Again, we introduce top and bottom tools. Let k again be some even integer. Define the two types of tools as follows. There are k top tools, each job requiring $k/2$

of them. There are $k/2 \times (k/2+1)$ bottom tools, of which each job requires $k/2 \times k/2$. The tool magazine capacity $C = (k/2)^2 + k$. We divide the bottom tools into $k/2+1$ sets, each consisting of $k/2$ tools. Each job requires all tools in all but one of these sets, and none of the bottom tools in the remaining set. We have a set of jobs for each possible choice of $k/2$ top tools out of k top tools. This set contains one job for each possible bottom tool requirement. Thus we have $\left(\begin{array}{c} k \\ k/2 \end{array} \right) \times (k/2+1)$ jobs (see Figure 7.3). Obviously, for the Batch Selection problem, an optimal batch is a set of $\left(\begin{array}{c} k \\ k/2 \end{array} \right)$ jobs with identical bottom tool requirements. Moreover, an optimal solution for the Job Grouping problem is to form $k/2+1$ groups (each of them an optimal batch), one for each possible bottom tools requirement.

Let us now study the behavior of heuristic 7. The heuristic must delete $k/2$ tools. We claim that the heuristic deletes (or may delete) $k/2$ top tools. Suppose after iteration $i, i \in \{0, \ldots, k/2-1\}$ the heuristic has not deleted bottom tools yet. Then the number of remaining jobs equals $\left(\begin{array}{c} k-i \\ k/2 \end{array} \right) \times (k/2+1)$. By symmetry, every bottom tool is required by a fraction of $\frac{k/2}{k/2+1}$ of all jobs. Similarly, every top tool is required by a fraction of

$$\frac{\left(\begin{array}{c} k-i-1 \\ k/2-1 \end{array} \right)}{\left(\begin{array}{c} k-i \\ k/2 \end{array} \right)} = \frac{k/2}{k-i} \leq \frac{k/2}{k/2+1}$$

of all jobs. Hence the heuristic may select a top tool again. After $k/2$ such iterations we thus end up with $k/2$ top tools and all bottom tools and a batch of $k/2+1$ jobs. Now, since $C < k^2$, this yields a ratio of

$$\Omega\left(\begin{array}{c} \sqrt{C} \\ \sqrt{C}/2 \end{array} \right) / \sqrt{C}$$

for the Batch Selection problem.

We now show that the heuristic performs equally bad on the job grouping problem. We show that when solving the job grouping problem by repeatedly appying heuristic 7 to form batches, we may get a batch for each possible top tool requirement. In view of the discussion above, it suffices to notice that for every set of jobs J such that J contains at least two jobs with distinct top tool requirements, there is always some top tool that is required by at most $\frac{k/2}{k/2+1}$ jobs. ■

The heuristic proposed by Dietrich Lee & Lee [1991] solves the previous instances optimally. We have, however, the following theorem :

Theorem 7.4 Heuristic 8 has worst case ratio of at least $\Omega(\left(\begin{array}{c} \sqrt{C} \\ \frac{\sqrt{C}}{2} \end{array} \right))$, and $\Omega(n)$.

$$\left(\begin{array}{cccccc|cccccc|cccccc}
1&1&1&0&0&0&1&1&1&0&0&0&1&1&1&0&0&0\\
1&0&0&1&1&0&1&0&0&1&1&0&1&0&0&1&1&0\\
0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\
0&0&1&0&1&1&0&0&1&0&1&1&0&0&1&0&1&1\\
\hline
1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0\\
1&1&1&1&1&1&0&0&0&0&0&0&1&1&1&1&1&1\\
1&1&1&1&1&1&0&0&0&0&0&0&1&1&1&1&1&1\\
0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1\\
0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1
\end{array}\right)$$

Figure 7.3: Tool job matrix of a worst case instance for heuristic 7, ($k = 4$)

	123	124	125	126	...	456
12	1	1	1	1		0
13	1	0	0	0		0
14	0	1	0	0		0
15	0	0	1	0		0
16	0	0	0	1		0
23	1	0	0	0		0
⋮						
56	0	0	0	0	...	1

Figure 7.4: The dummy tool job matrix D for $m = 6$.

Proof. Let m be an even integer. First, we construct a dummy tool job matrix D as follows. For each possible subset of cardinality $m/2$ of the integers $1, \ldots, m$, we have a column in D. Each row of D corresponds to a 2-element subset of the integers $1, \ldots, m$. Entry $d_{i,j} = 1$ if the i-th 2-element subset is contained in the j-th $(m/2)$-element subset, and zero otherwise (see Figure 7.4).

The jobs corresponding to the columns of D satisfy the following property (P) :

Let p_j, p_k and p_l be three jobs. There exists a row i of D such that $p_{ij} = 1, p_{ik} = p_{il} = 0$.

From the matrix D we derive an instance that yields the desired ratio as follows. Again we introduce top tools and bottom tools. There are two sets of bottom tools, called bottom sets, each job requires all the tools in one of the bottom sets, and none

of the other bottom tools. Each set of bottom tools consists of

$$\binom{m}{2} - \binom{m/2}{2}$$

tools. The top tools correspond to the tools as defined by the matrix D. Moreover for each column of D we have one job for each of the aforementioned bottom sets. Thus we have $\binom{m}{m/2} \times 2$ jobs. We set $C = 2\binom{m}{2} - \binom{m/2}{2}$. Then the heuristic proposed by Dietrich, Lee & Lee may pick batches in which all bottom tools are required and only $\binom{m/2}{2}$ top tools as follows. It selects an arbitrary job to begin with. All jobs whose top tool requirements and bottom tool requirements differ from the requirements of the already selected job, can not be added to the batch : the batch would require more than C tools. The single job that has the same top tool requirements, but requires the other set of bottom tools, has weight zero. Moreover, because of (P), all jobs that have the same bottom tool requirements, but different top tool requirements, also have weight zero. Thus, the rule may select the single job with the same top tool requirements, filling up the tool magazine completely. In this way the rule selects groups of size 2, whereas the optimal solution contains 2 groups of size $\binom{m}{\frac{m}{2}}$. Since $C = \Theta(m^2)$ this yields the desired ratio. ∎

The results in the previous discussion being discouraging, the question arises whether polynomial approximation algorithms with better worst case ratios exist. The relationship between Clique and Batch Selection and the relationship between Job Grouping and Set Covering, that was informally established in the previous section, suggest that good approximation algorithms may not exist : both Clique and Set Covering are notoriously hard to approximate. It has long been open whether a polynomial approximation algorithm for Clique with a constant worst case ratio could exist. It was well known however, see Garey & Johnson [1978], that the existence of a polynomial approximation algorithm for Clique would imply the existence of a polynomial approximation scheme for this problem, which was regarded to be unlikely to exist. In this section we obtain exactly the same results for Batch Selection. Only recently, Arora et al. [1993] have shown that a polynomial approximation scheme for Clique cannot exist unless P = NP. They even improved on this result by showing that there is some ϵ such that there cannot exist an $O(m^\epsilon)$ approximation algorithm for Clique (where m is the number of nodes) unless P = NP.

The same breakthrough that led to the negative result regarding the approximability of Clique, enabled Lund & Yannakakis [1993] to show that for any $d, 0 < d < \frac{1}{4}$, a polynomial approximation algorithm for Set Covering with worst case ratio $d \log n$ cannot exist, unless NP is contained in $DTIME[n^{poly \log n}]$ (Lund & Yannakakis [1993]). In this section, we briefly discuss the implications of this result for Job Grouping. We also establish a relationship between Job Grouping and Tool Switching.

A first negative result is the following theorem, in the spirit of Theorem 6.12 in Garey & Johnson [1978] :

Theorem 7.5 Either the Batch Selection Problem can be solved by a polynomial time approximation scheme, or else there is no polynomial time approximation algorithm with constant worst case ratio for this problem.

Proof. Suppose that H is a polynomial time approximation algorithm with finite worst case ratio r. Let I be an instance of Batch Selection. For any ϵ let l_ϵ be the smallest integer such that $r^{\frac{1}{l_\epsilon}} < 1 + \epsilon$. We construct an approximation scheme S that delivers a solution with value $S(I)$ for I, such that $OPT(I)/S(I) \leq 1 + \epsilon$. Moreover its running time depends polynomially on n, m and $\log C$ and on the running time of H, as required, and is exponential in l_ϵ.

To obtain a result as mentioned in the theorem, we need a method to 'square' an instance. In case of the Clique problem, a squaring mechanism is readily available, see Garey & Johnson [1978]. In case of Batch Selection the squaring operator is a little more artificial. What we do exactly, is the following. Given an instance of Batch Selection, consisting of a magazine capacity C and a tool job matrix A, squaring gives in polynomial time a new instance of Batch Selection, with capacity C' and tool-job matrix A' such that

(1). From any solution of I' with value $s'(I')$ to I' we can construct a solution of I with value $s(I)$ such that $s(I)^2 \geq s'(I')$.

(2). Moreover, $OPT(I') = OPT(I)^2$.

Notice that this suffices to prove the Theorem. Given an instance of Batch Selection, we square $\lceil \log(l_\epsilon) \rceil$ times, we then apply H and can construct a solution for the original problem instance of value at least $OPT(I)/(1 + \epsilon)$, by definition of l_ϵ.

We construct C' and A' from C and A as follows. The squared magazine capacity $C' = (C + 3)C$. For each column of A we get n columns in A'. For each row of A we get $C + 3$ rows in A'. Column $a_{(j-1)n+l}$ depends on the columns a_j and a_l. To be precise, the transformation is as follows :

(1). if $a_{i,j} = 1$, then $a'_{i,(j-1)n+l} = 1$, for all $i = 1, \ldots, m$, $j, l = 1, \ldots, n$;

(2). if $a_{i,l} = 1$, then $a'_{(C+2)m+i,(j-1)n+l} = 1$, for all $i = 1, \ldots, m$, $j, l = 1, \ldots, n$;

(3). if $a_{i,j} = 1$ or $a_{i,l} = 1$, then $a'_{m+(C+1)(i-1)+k,(j-1)n+l} = 1$, for $k = 1, \ldots, C + 1, i = 1, \ldots, m$, $j, l = 1, \ldots, n$.

(4). $a'_{ij} = 0$ otherwise.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Figure 7.5: Batch Selection Instance I, $C = 2$, A

$$\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
 & & & & & & & & \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
 & & & & & & & & \\
1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1
\end{pmatrix}$$

Figure 7.6: The instance $I' = I^2$, $C' = 10$, $A' = A^2$.

An example of this transformation is given in Figures 5 and 6. Matrix A' may be interpreted as follows. For each ordered pair of jobs (p_j, p_l) in I, we obtain a job, $p'_{(j-1)n+l}$ in I'. Further, there are three sets of tools. One set, to be referred to as top tools, consists of tools $t'_1, \ldots, t'_m$ and duplicates the tool requirements of p_j. Another set consists of tools $t'_{m(C+2)+1}, \ldots, t'_{m(C+2)+m}$, duplicates the tool requirements of job p_l. The tools in this second set will be referred to as bottom tools. The third set of tools depends on both p_j and p_l. Indeed if some tool t_i is required by either p_j or p_l, then the tools $t'_{m+(C+1)(i-1)+k}, k = 1, \ldots, C+1$ are required for $p'_{(j-1)n+l}$ and $p'_{(l-1)n+j}$.

It follows from this description that if a job requires tool $t'_{m+(C+1)(i-1)+p}$, for some $1 \leq i \leq n, 1 \leq p \leq C+1$ then it requires $t'_{m+(C+1)(i-1)+q}$, for all $1 \leq q \leq C+1$. Thus, in any solution, it is pointless to have $r, 0 < r < C+1$ of the tools $t'_{m+(C+1)(i-1)+q}, 1 \leq q \leq C+1$. In the following we may therefore assume, without loss of generality, that a solution contains either all or none of the tools $t'_{m+(C+1)(i-1)+q}, 1 \leq q \leq C+1$, for all i.

Consider again some job A_j in I', i.e. some column of A'. The definition of A' implies that for all $i, 1 \leq i \leq m$, if $a_{i,j} = 1$, then $a_{m+(C+1)(i-1)+q,j} = 1$ for all $1 \leq q \leq C+1$. We conclude again that there is no benefit in selecting tool t_i in a solution unless all tools $t'_{m+(C+1)(i-1)+q}, 1 \leq q \leq C+1$ are selected. Similarly, if a job requires tool $t'_{(C+2)m+i}$, then it also requires tools $t'_{m+(C+1)(i-1)+q}$, for all $1 \leq q \leq C+1$. Again, there is no use in selecting tool $t'_{(C+2)m+i}$, unless tools $t'_{m+(C+1)(i-1)+q}$ are selected, for all $1 \leq q \leq C+1$. For convenience, we call *block* i $(1 \leq i \leq m)$, the set of tools $t'_{m+(C+1)(i-1)+q}$ where $1 \leq q \leq C+1$. We call t'_i the *corresponding top tool*, and call $t'_{m+(C+1)m+i}$ the *corresponding bottom tool.*

The above observations lead us to the conclusion that if we select r blocks, then we select at most r top tools and at most r bottom tools. Since $C' = (C+3)C$, and each block consists of $C+1$ tools, this implies that we may as well select at least C blocks. Let us first consider the case where we select at least $C+1$ blocks. Since $C' = (C+3)C < (C+2)(C+1)$, we cannot select $C+2$ blocks. Hence suppose we select $C+1$ blocks. This means we can select $C(C+3) - (C+1)(C+1) = C-1$ top and bottom tools altogether. But this in turn implies that we have selected at least two blocks for which there are no corresponding top and bottom tools. Hence, we can unselect these tools without reducing the number of jobs in the batch, which brings us again in the situation where the solution contains only $(C-1)$ blocks.

This leaves us with the case in which we have selected C blocks. In this case, we can select $2C$ top and bottom tools altogether. We know however, that there is no benefit in selecting top and bottom tools whose corresponding block is not selected. Hence, given a selection of C blocks, we may assume that the remaining tools in the solution are the corresponding top and bottom tools.

Now that we have imposed some structure on the sets of tools that are selected, let us check the implications for the number of jobs in the corresponding batch. Consider a solution S to I, i.e. a set of selected tools $t_{i_1}, \ldots, t_{i_C}$, with value $s(I)$ and let p_1 to p_k be the jobs of I that can be performed using $t_{i_1}, \ldots, t_{i_C}$. We construct a solution to I' with value $s'(I')$ by selecting the top tools $t'_{i_1}, \ldots, t'_{i_C}$ as well as the corresponding blocks and bottom tools. It can be seen as follows that $s'(I') = s(I)^2$. Solution S'

allows exactly the jobs of I' corresponding to all ordered pairs (p_j, p_l) , $j, l = 1, \ldots, k$ to be performed.

Conversely, given a solution S' of value $s'(I')$ to I' which requires (without loss of generality) C blocks and their corresponding top and bottom tools, we obtain a solution to I of value $\sqrt{(s'(I')}$ by selecting tool t_i in I, only if t'_i in I' is selected, $1 \leq i \leq m$. Let $t'_{i_1}, \ldots, t'_{i_C}$, denote the top tools used in the solution S' to I', and let $p_1, \ldots, p_k$ be the jobs of I that can be performed using $t_{i_1}, \ldots, t_{i_C}$: Solution S' contains exactly those jobs of I' corresponding to all ordered pairs (p_j, p_l) for which p_j and p_l are in the batch corresponding to S. Hence, $k = \sqrt{s'(I')}$. ∎

For the Job Grouping problem we have the following negative result on its approximability :

Theorem 7.6 For any $d < \frac{1}{4}$, the Job Grouping Problem cannot be approximated within a factor of $d \log n$ in polynomial time unless NP is contained in $DTIME[n^{poly \log n}]$, even if $C = m - 1$.

Proof. Lund & Yannakakis [1993] prove that for any $d < \frac{1}{4}$, Set Covering can not be approximated within a factor of $d \log k$ in polynomial time unless NP is contained in $DTIME[n^{poly \log n}]$, where k is the number of rows of the Covering Problem. Hence we prove simply by giving a mapping from Set Covering to the special case of Job Grouping where $C = m - 1$ and $n = k$. Notice that in such instances the Batch Selection problem can be solved trivially, so that finding a minimal cover, or partition, indeed poses the only difficulty.

Let $s_1, \ldots, s_p$ be the subsets in the Set Covering instance, and let $e_1, \ldots, e_q$ be the elements of its ground set. Then the covering matrix B has $b_{ij} = 1$ if s_j contains e_i and zero otherwise. The following construction of a tool job matrix A is due to Crama & Oerlemans [1992]. We set $a_{ij} = 1 - b_{ji}$ for all $1 \leq i \leq p, 1 \leq j \leq q$. Setting $m = p$, and $C = p - 1$ yields, together with A an instance of Job Grouping. One checks that if some subset s_i contains the two elements e_k and e_l, then both columns A_k and A_l have a zero in row i. Since the tool magazine $C = m - 1$ this means that jobs j_k and j_l may be in a same batch. More generally, every subset in the Set Covering problem corresponds to a batch in the Job Grouping problem, and every maximal batch is a subset in the Set Covering problem. ∎

We close this section by discussing the relationship between Job Grouping and Tool Switching. Notice again that both problems require the same data. Now let $v_{TS}(S)$ be the value of solution S to the tool switching problem. The same switches imply a solution S' for Job Grouping of value $v_{JG}(S') \leq v_{TS}(S)$. Conversely, let $v_{JG}(S')$ be the value of any solution S' to Job Grouping. Since between each two groups we can have at most C switches, there is a solution S to Tool Switching of value $v_{TS}(S) \leq C \times v_{JG}(S')$. Thus we also have that :

$$\frac{1}{v_{TS}(OPT_{TS})} \leq \frac{1}{v_{JG}(OPT_{JG})} \leq \frac{C}{v_{TS}(OPT_{TS})}.$$

If we have an approximation algorithm H with worst case ratio r_1 for Job grouping giving solution s', then we can construct a solution S for Tool Switching, and it holds that

$$r_1 \geq \frac{v_{JG}(S')}{v_{JG}(OPT_{JG})} \geq \frac{v_{JG}(S')}{v_{TS}(OPT_{TS})} \geq \frac{v_{TS}(S)/C}{v_{TS}(OPT_{TS})}.$$

Thus, H yields a Cr_1 approximation algorithm for Tool Switching.

If we have an approximation algorithm H' for Tool Switching with worst case ratio r_2, giving solution S, then we can construct a solution S' of Job Grouping so that,

$$r_2 \geq \frac{v_{TS}(S)}{v_{TS}(OPT_{TS})} \geq \frac{v_{JG}(S')}{v_{TS}(OPT_{TS})} \geq \frac{v_{JG}(S')/C}{v_{JG}(OPT_{JG})}.$$

Thus H' yields an approximation algorithm with worst case ratio Cr_2 for Job Grouping.

7.4 Further research

In combination, the results of Sections 2 and 3 still leave a rather big gap between the ratio the best approximation algorithms achieve and what is likely to be unattainable. The negative results of Section 3 are expressed in constants and n the number of jobs, in which case a superpolynomial bound on the worst ratio cannot be obtained. However, we have seen that, expressed in terms of the tool magazine C, superpolynomial worst case ratios are possible. It might well be the case that there is some $\epsilon > 0$, such that unless $P = NP$ approximation algorithms with worst case ratio smaller than or equal to 2^{C^ϵ} cannot exist for both the Job Grouping and the Batch Selection problem. The possibility to achieve superpolynomial worst case results hopefully provides an extra challenge in investigating this possibility.

On the positive side, Crama & van de Klundert [1994] show that iteratively generating batches with an α approximation algorithm for the batch selection problem until all jobs are in some batch, yields an approximation algorithm for Job Grouping with approximation ratio of $O(\alpha \log(n/\alpha))$. Chapter 8 discusses these results.

In view of this result, observe that a strengthening of the negative result regarding Job Grouping has its implication for what is likely to be attainable for Batch Selection. For example, if one could show that there is some ϵ such that there cannot exist a polynomial approximation algorithm with worst case ratio n^ϵ, unless P = NP, then there could not exist a polynomial approximation algorithm for Batch Selection with worst case ratio strictly smaller than n^ϵ unless P = NP. Lund & Yannakakis [1993] have obtained results for problems that are related in a similar fashion. For example, they were able to show that Graph Coloring cannot be approximated within a subpolynomial ratio, as is the case for its generating subproblem Independent Set (Arora et al. [1993]).

Chapter 8

Approximation algorithms for integer covering problems via greedy column generation

8.1 Introduction

Many combinatorial optimization problems can be formulated as covering problems of the form:

$$\begin{array}{lrl} \text{minimize} & cx & \\ \text{s.t.} & Ax & \geq b \\ & x & \in \mathbb{Z}_+^n \end{array} \qquad (P)$$

where $c \in \mathbb{R}_+^n$, $A \in \mathbb{Z}_+^{m \times n}$ and $b \in \mathbb{Z}_+^m$. (Notice that all data are nonnegative.)

In some cases, this covering formulation is not polynomial in the input size of the original problem. Typically, for instance, m may be a parameter of the original problem, whereas n is exponential in m. This can be illustrated by the covering formulation of the Cutting Stock Problem due to Gilmore and Gomory [1964] (see also Chvátal [1979]; for a general discussion of Cutting Stock Problems see Dyckhoff [1990], Haessler and Sweeney [1991]). In this problem there are rolls of material of various length, called raws, from which pieces of specified length, called finals, have to be cut. The demand for each final is given. The cost of cutting finals from a raw may depend on the size of the raw, and/or on the cutting pattern. A covering formulation can be obtained as follows. Each row corresponds to some final. Each column corresponds to a feasible cutting pattern, i.e. to some way of cutting certain finals from some specific raw. The cost of pattern j is denoted c_j, the number of finals of type i produced by pattern j is denoted a_{ij}, and the demand for final i is denoted b_i. It is not hard to see that the number of columns might be exponential in the problem size.

Gilmore and Gomory [1964] proposed a column generation technique to solve the linear relaxation of such large-scale models. In their approach, the columns of (P) are not explicitly listed. Rather, the pricing step of the simplex algorithm is implemented as follows : given current values $u_1, \ldots, u_m$ of the dual variables, the column generation subproblem $\max\{\Sigma_i u_i a_{ij} - c_j | j = 1, \ldots, n\}$ is solved in order to detect columns with positive reduced cost. For the method to be applicable, the *column generation problem* should be (relatively) easy to solve, and should not require a complete enumeration of all columns of (P). For instance, Minoux [1987] observed that the approach allows to solve the linear relaxation of (P) in polynomial time whenever the column generation subproblem itself is polynomially solvable. When (P) is a Cutting Stock Problem, the column generation subproblem turns out to be a knapsack problem, which is NP-hard, but can be solved reasonably fast in practice (see Gilmore and Gomory [1964] or Chvátal [1979] for details). Similar approaches have been successfully applied to many other problems.

The research in this chapter was motivated by the Job Grouping Problem of Chapter 7. In the Job Grouping Problem there is set of jobs, each of which requires tools for its processing. A machine that processes a job must have all the tools required by this job in its tool magazine. In a basic model, there is a tool magazine capacity C, and each tool requires one position (capacity unit) in the magazine. A group is a set of jobs that altogether do not require more than C tools (and hence have the attractive property that they can be processed on a machine without being interrupted for tool changes). The Job Grouping Problem is now to find a minimum cardinality partitioning of the set of jobs into groups. A covering formulation for the Job Grouping Problem is readily available (see also Theorem 7.6) Each row of A corresponds to a job, and each (of the possibly exponetially many) column to a group. All groups have unit cost, and the right hand sides are also equal to one.

In Chapter 7, we were interested in finding approximation algorithms for the Job Grouping Problem with good worst case performance. In the current chapter, we continue the search by studying a class of polynomial time approximation algorithms that can be applied to the Job Grouping Problem, but to several other combinatorial problems, such as the Cutting Stock Problem, as well.

Thus, rather than solving the linear relaxation of (P) (which only yields a lower-bound on the optimal value of (P)), this chapter addresses the issue of computing a good heuristic solution of (P). More precisely, we are going to discuss a class of approximation algorithms for problem (P) and to derive bounds on their worst case ratio (the worst case ratio of an approximation algorithm is the supremum, taken over all problem instances, of the ratio between the value of the solution delivered by the approximation algorithm and the optimal value of the problem instance; see e.g. Nemhauser and Wolsey [1988]). As customary when discussing heuristics, we will be especially interested in polynomial time algorithms.

Our algorithms build on the algorithm Greedy studied by Johnson [1974], Lovász [1975] (in the case where all entries of b and c are 1), Chvátal [1979] (in the case where all entries of b are 1) and Dobson [1982] (for general b and c). Dobson [1982] showed that

Greedy has worst case ratio $H(A_{max})$ where $A_{max} = \max_{1 \leq j \leq n} \sum_{i=1}^{m} a_{ij}$, and $H(d) = \sum_{i=1}^{d} \frac{1}{i}$. It is well known that $\ln d < H(d) < \ln d + 1$. (Recently Lund and Yannakakis [1993] proved that there cannot exist an approximation algorithm for the Set Covering Problem having worst case ratio $c \times \log m$ with $0 < c < 1/4$, unless P = NP.) The previous results assume that the formulation (P) is explicitly given as input. Our goal will be to circumvent the difficulties posed when this is not the case, i.e. when n is very large.

For instance, Greedy requires to select (iteratively) a column j of A which minimizes the ratio $c_j / \sum_i a_{ij}$. Of course, this step poses no particular difficulty if all columns of A are explicitly available. When this is not the case, it is also clear that the column to be selected can be found by solving a generation subproblem similar to the one discussed above. If this subproblem is polynomially solvable, then the complexity of Greedy is only affected by a polynomial factor. However, when the subproblem itself is NP-hard, Greedy may no longer be a polynomial-time procedure (for instance, in the framework of Gilmore and Gomory's cutting stock problem, the subproblem is a knapsack problem, as before). In this chapter we concentrate on the situation where an approximation algorithm with worst case ratio α is available to solve the column generation subproblem. (Notice that α may not be a constant, and that the approximation algorithm need not be polynomial - though this is obviously the most attractive case). In other words, we assume that we can find (in polynomial time) a column k of (P) such that

$$c_k / \sum_i a_{ik} \leq \alpha \times \min_{1 \leq j \leq n} \{c_j / \sum_i a_{ij}\}$$

where $\alpha \geq 1$ (this extends to problem (P) the Master-Slave approach described by Simon [1990] for 'classical' set covering problems). Under this assumption, we prove in Section 2 that the worst case ratio of a modified version of Greedy (called αGreedy) is α times the worst case ratio of Greedy (Theorem 8.1). In the special case where all entries of b and c are 1, Simon [1988] established the worst case ratio $\alpha \ln m$ for αGreedy. Theorem 8.1 refines this result and extends it to general values of b and c. As a matter of fact, under Simon's assumptions, we can even obtain a slightly sharper result, which we state as Theorem 8.2.

When applied to (P), αGreedy may require the selection of $O(n + m)$ columns. Therefore, the complexity of αGreedy is $O((n + m)T)$, where T is (up to some algebraic manipulations on A, b and c) the running time of the approximation algorithm used to solve the column generation subproblem. Notice that this running time is not satisfactory when n is large (i.e., exponential in m), *even* if we assume that T is polynomial in m. To remedy this difficulty, we propose in Section 3 a modification of αGreedy that reduces its complexity to $O(mT)$ while degrading its worst case ratio by a factor of at most 2 (see Theorem 8.3). In particular, the resulting HyperαGreedy algorithm runs in polynomial time if T is polynomial in m. Section 4 discusses applications of the derived results.

8.2 αGreedy algorithms and their worst case ratio.

In this section we first describe the greedy algorithm for problem (P) (Dobson [1982]). Thereafter we present the αGreedy algorithm and we derive its worst case ratio. We give a worst case example, showing that the bound is tight. Finally, we derive a sharper result for the case where all entries of b and c are 1.

The idea behind the greedy algorithm is, in each iteration, to increase the variable corresponding to the relatively cheapest column. Hence, the greedy algorithm picks a column k such that

$$k = \arg\min_j (c_j / \sum_{i=1}^{m} a_{ij}),$$

and increases x_k by 1 unit. In order for the greedy algorithm not to be misled by extremely high values of the constraint coefficients, we assume as in Dobson [1982] that $a_{ij} \leq b_i$ for all i in $\{1,\ldots,m\}$, j in $\{1,\ldots,n\}$ (otherwise the data should be updated accordingly.) We now formulate the greedy algorithm in a slightly different version from the one in Dobson [1982].

Here and in the remainder of this chapter, we implicitly assume that any row or column which becomes zero during the execution of the algorithm is left out of consideration in all subsequent iterations.

This presentation of the algorithm facilitates its worst case analysis. However, it is easy to see that the inner-while loop could also be replaced by a statement of the form

$$x_k := x_k + \min_{1 \leq i \leq m} \lfloor b_i / a_{ik} \rfloor$$

with corresponding updates of b and z. This would result in an algorithm with $O(n+m)$ iterations.

We now give the description of the αGreedy algorithm. The idea here is that instead of iteratively picking the relatively cheapest column, we pick some column k such that

$$c_k / \sum_{i=1}^{m} a_{ik} \leq \alpha \times \min_{1 \leq j \leq n} c_j / \sum_{i=1}^{m} a_{ij}, \tag{8.1}$$

where $\alpha \geq 1$, and α is fixed throughout the algorithm. Formally the αGreedy algorithm is:

Clearly αGreedy can be modified in the same way as Greedy, so as to reduce its number of iterations to $O(n+m)$. In particular, αGreedy can be implemented to run in polynomial time if there is a polynomial algorithm to select a column k satisfying (1).

As mentioned in the introduction, Dobson [1982] proved that Greedy has worst case ratio $H(A_{max})$. We claim the following worst case ratio for αGreedy :

Algorithm 1 : Greedy

$x := 0$
$z := 0$
while $b \neq 0$ *do*
begin

$k := \arg\min_{1 \leq j \leq n} (c_j / \sum_{i=1}^{m} a_{ij})$

while for all i, $b_i \geq a_{ik}$ *do*
begin
$x_k := x_k + 1$
$b_i := b_i - a_{ik}$ *for all* i
$z := z + c_k$
end
$a_{ij} := \min(a_{ij}, b_i)$ *for all* i, j
end.

Figure 8.1: Algorithm Greedy

Theorem 8.1 If x^* is an optimal solution of problem (P) and x' is the solution computed by αGreedy, then

$$cx'/cx^* \leq \alpha \times H(A_{max})$$

and this bound is tight.

Proof. Since the proof is analogous to the proof in Dobson [1982] for the worst case ratio of Greedy, we introduce the same notations and we skip most of the parts that go through with little or no modification. We consider one execution of the body of the inner while statement to be one iteration of αGreedy and we denote by t the total number of iterations. During the execution of the algorithm, A and b change frequently. The notations $A^r = (a_{ij}^r)$ and $b^r = (b_i^r)$ refer to the data at the beginning of iteration $r, r = 1, \ldots, t+1$, with $A^{t+1} = 0$ and $b^{t+1} = 0$, by definition. We also let $w_j^r = \sum_{i=1}^{m} a_{ij}^r$ for all j in $\{1, \ldots, n\}$. Notice that the following implication holds, in view of the description of αGreedy: for $i = 1, \ldots, m$, $j = 1, \ldots, n$ and $r = 1, \ldots, t$

$$\text{if } a_{ij}^r < a_{ij}^1 \text{ then } a_{ij}^r = b_i^r \tag{8.2}$$

Algorithm 2 : αGreedy

$x := 0$
$z := 0$
while $b \neq 0$ *do*
begin

select k *such that*
$c_k / \sum_{i=1}^{m} a_{ik} \leq \alpha \times \min_{1 \leq j \leq n}(c_j / \sum_{i=1}^{m} a_{ij})$

while for all i, $b_i \geq a_{ik}$ *do*
begin
$x_k := x_k + 1$
$b_i := b_i - a_{ik}$ *for all* i
$z := z + c_k$
end
$a_{ij} := min(a_{ij}, b_i)$ *for all* i, j
end.

Figure 8.2: Algorithm αGreedy

We also use a set of step functions, one for each constraint of (P). Let k_r be the column picked in iteration r of αGreedy. Then we define $p_i(s)$ to be the function with domain $[0, b_i)$ and such that

$$p_i(s) = c_{k_r}/w^r_{k_r} \text{ if } s \in [b_i^{r+1}, b_i^r) \text{ for } r = 1, \ldots, t.$$

Finally we introduce a step function p_{ij} for each a_{ij}, where

$$p_{ij}(s) = \begin{cases} c_j/w_j^r & \text{if } s \in [a_{ij}^{r+1}, a_{ij}^r) \text{ for } r = 1, \ldots, t \\ p_i(s) & \text{if } s \in [a_{ij}^1, b_i). \end{cases}$$

The proof will be based on the following three lemmas.

Lemma 8.1 If x' is the solution computed by the αGreedy algorithm, then

$$cx' = \sum_{i=1}^{m} \int_0^{b_i} p_i(s)ds.$$

Proof. See Dobson [1982]; the reasoning goes through without modification. ∎

Lemma 8.2 For all $i = 1, \ldots, m$ and $j = 1, \ldots, n$,

$$\frac{a_{ij}}{b_i} \int_0^{b_i} p_i(s)ds \leq \alpha \int_0^{a_{ij}} p_{ij}(s)ds.$$

Proof. Fix $j \in \{1, \ldots, n\}$ and $i \in \{1, \ldots, m\}$. The inequality holds if $a_{ij} = 0$. So assume $a_{ij} \neq 0$.
Define $Min = \min_{0 \leq s < a_{ij}} p_{ij}(s)$ and $Max = \max_{a_{ij} \leq s < b_i} p_{ij}(s)$. Let r be such that

$$Min = \frac{c_j}{w_j^r} \equiv p_{ij}(a_{ij}^{r+1}),$$

where $a_{ij}^{r+1} < a_{ij}$, and let u and $k = k_u$ be such that

$$Max = \frac{c_k}{w_k^u} \equiv p_{ij}(b_i^{u+1}),$$

where $b_i^{u+1} \geq a_{ij}$.
Notice that from (1), it follows that $a_i^{r+1} = b_i^{r+1}$, and hence $r > u$. We now claim that the following inequalities hold :

$$\alpha \times Min = \alpha \frac{c_j}{w_j^r} \geq \alpha \frac{c_j}{w_j^u} \geq \frac{c_k}{w_k^u} = Max. \tag{8.3}$$

Indeed, by definition,

$$w_j^r = \sum_{i=1}^{m} a_{ij}^r \text{ and } w_j^u = \sum_{i=1}^{m} a_{ij}^u.$$

Since $u < r$ it must be the case that $a_{ij}^r \leq a_{ij}^u$ for all $i \in \{1, \ldots, m\}$. This proves the first inequality in (3). The second inequality is immediate by definition of αGreedy.
Now Lemma 8.2 can be proved as follows :

$$\begin{aligned}
\frac{1}{b_i} \int_0^{b_i} p_i(s)ds &= \frac{1}{b_i}[\int_0^{a_{ij}} p_i(s)ds + \int_{a_{ij}}^{b_i} p_i(s)ds] \\
&\leq \frac{1}{b_i}[\int_0^{a_{ij}} \alpha p_{ij}(s)ds + \int_{a_{ij}}^{b_i} p_{ij}(s)ds] \quad \text{(by definition of } \alpha\text{Greedy)} \\
&\leq \frac{1}{b_i}[\int_0^{a_{ij}} \alpha p_{ij}(s)ds + (b_i - a_{ij})Max] \\
&\leq \frac{1}{b_i}[\int_0^{a_{ij}} \alpha p_{ij}(s)ds + (b_i - a_{ij})\alpha \times Min] \quad \text{(by (8.3))} \\
&\leq \frac{\alpha}{b_i}[\int_0^{a_{ij}} p_{ij}(s)ds + \frac{b_i - a_{ij}}{a_{ij}} \int_0^{a_{ij}} p_{ij}(s)ds] \\
&= \frac{\alpha}{a_{ij}} \int_0^{a_{ij}} p_{ij}(s)ds. \qquad \blacksquare
\end{aligned}$$

Lemma 8.3 For all $j = 1, \ldots, n$,

$$\sum_{i=1}^{m} \frac{a_{ij}}{b_i} \int_0^{b_i} p_i(s)ds \leq \alpha c_j h_j$$

$$\text{where } h_j = \sum_{r=1}^{v_j} \frac{w_j^r - w_j^{r+1}}{w_j^r} \text{ and } v_j = \min\{r | w_j^{r+1} = 0\}.$$

Proof. One proves, as in Dobson [1982], that

$$\sum_{i=1}^{m} \int_0^{a_{ij}} p_{ij}(s)ds = c_j h_j$$

Together with Lemma 8.2, this easily implies the statement. ∎

Now we are in a position to prove Theorem 8.1. Let x be any feasible solution, then the following holds for $i = 1, \ldots, m$:

$$\sum_{j=1}^{n} \frac{a_{ij} x_j}{b_i} \geq 1 \text{ and hence :}$$

$$\begin{aligned}
cx' &= \sum_{i=1}^{m} \int_0^{b_i} p_i(s)ds && \text{(by Lemma 8.1)} \\
&\leq \sum_{i=1}^{m} \sum_{j=1}^{n} \frac{a_{ij} x_j}{b_i} \int_0^{b_i} p_i(s)ds \\
&= \sum_{j=1}^{n} (\sum_{i=1}^{m} \frac{a_{ij}}{b_i} \int_0^{b_i} p_i(s)ds) x_j \\
&\leq \alpha \sum_{j=1}^{n} (c_j h_j) x_j && \text{(by Lemma 8.3)} \\
&\leq \alpha \max_{1 \leq j \leq n} (h_j) \sum_{j=1}^{n} c_j x_j.
\end{aligned}$$

Hence we have

$$\frac{cx'}{cx} \leq \alpha \max_{1 \leq j \leq n} h_j,$$

where x is any feasible solution. Dobson [1982] shows that

$$\max_{1 \leq j \leq n} h_j \leq H(A_{max}).$$

This inequality yields the desired upper bound.

The following problem instance (adapted from Chvátal [1979]) shows that this bound is tight for any $\alpha \geq 1$, and any (integer) value of A_{max}.

$$
\begin{array}{lllllll}
\text{Min} & \frac{\alpha}{d}x_1 & +\frac{\alpha}{(d-1)}x_2 & +\cdots+ & \frac{\alpha}{2}x_{d-1} & +\alpha x_d & +x_{d+1} \\
\text{s.t.} & x_1 & & & & & +x_{d+1} \geq 1 \\
 & & x_2 & & & & +x_{d+1} \geq 1 \\
 & & & \ddots & & & \vdots \\
 & & & & x_{d-1} & & +x_{d+1} \geq 1 \\
 & & & & & x_d & +x_{d+1} \geq 1.
\end{array}
$$

The optimal solution of this instance is given by $x_i^* = 0 (1 \leq i \leq d)$, $x_{d+1} = 1$, and has value 1. But αGreedy could set $x_i = 1$ in iteration $i (1 \leq i \leq d)$, thus resulting in a solution with value $\alpha H(d)$. ■

Notice that the worst case instance is in fact a Weighted Set Covering Problem. Since in the Weighted Set Covering Problem the constraint matrix is a 0-1 matrix, the running time of an αGreedy algorithm is $O(m)$ instead of the $O(n+m)$ in the case of integral constraints.

When all entries of b and c are 1 (Unweighted Set Covering Problem), we can obtain a sharper result than Theorem 8.1. (Theorem 8.2 strengthens a result of Simon [1990] - see also Johnson [1974b], Simon [1988] - who establishes the asymptotic worst case ratio $\alpha \ln m$ for αGreedy.)

Theorem 8.2 If x^* is an optimal solution of the Unweighted Set Covering Problem and x' is the solution given by αGreedy then

$$cx'/cx^* \leq \alpha H(\lceil d/\alpha \rceil) + \frac{d}{\lceil d/\alpha \rceil} - \alpha$$

and this bound is tight.

Proof. As in Lovász [1975], one proves in case of αGreedy that :

$$
\begin{aligned}
cx'/cx^* &\leq \sum_{i=1}^{\lceil d/\alpha \rceil - 1} \frac{\lfloor i\alpha \rfloor}{i(i+1)} + \sum_{i=\lceil d/\alpha \rceil}^{d-1} \frac{d}{i(i+1)} + 1 \\
&\leq \sum_{i=1}^{\lceil d/\alpha \rceil - 1} \frac{i\alpha}{i(i+1)} + d \times \sum_{i=\lceil d/\alpha \rceil}^{d-1} \frac{1}{i(i+1)} + 1
\end{aligned}
$$

$$= \alpha H(\lceil d/\alpha \rceil) - \alpha + d \times \sum_{i=\lceil d/\alpha \rceil}^{d-1} (\frac{1}{i} - \frac{1}{i+1}) + 1$$

$$= \alpha H(\lceil d/\alpha \rceil) + \frac{d}{\lceil d/\alpha \rceil} - \alpha$$

This bound can only be tight for integral values of α, as is easily checked. Hence we show now how to construct a worst case instance for arbitrary integral values for α and d. The worst case instance consists of $\lceil d/\alpha \rceil + 1$ groups of columns. The columns of the last group will constitute the optimal solution. The columns of the other groups will together constitute the solution found by αGreedy. For the construction of the columns we refer to the example. We give here the number of columns in each group, showing that the bound is tight.

In total there are $d \times (\lceil d/\alpha \rceil)!$ rows. Every column in the last group covers d rows and there are $(\lceil d/\alpha \rceil)!$ such columns. Every column of group i ($i = 1 \ldots \lceil d/\alpha \rceil$) covers i rows. In the *i-th* group ($i = 1 \ldots \lceil d/\alpha \rceil - 1$) there are $\alpha/i \times (\lceil d/\alpha \rceil)!$ columns. In group $\lceil d/\alpha \rceil$ there are $(d - \alpha(\lceil d/\alpha \rceil - 1)) \times (\lceil d/\alpha \rceil)!)/\lceil d/\alpha \rceil)$ columns. Any two columns in the last group are disjoint, as are any two columns not in the last group. Thus, together, the columns in groups $1 \ldots \lceil d/\alpha \rceil$ cover all $d \times (\lceil d/\alpha \rceil)!$ rows.

Together, all columns in group i intersect an arbitrary column of the last group in exactly α rows, for ($i = 1 \ldots \lceil d/\alpha \rceil - 1$). Similarly, all columns in group $\lceil d/\alpha \rceil$ intersect an arbitrary column of the last group in $(d - \alpha(\lceil d/\alpha \rceil - 1))$ rows, i.e. at most α rows. It follows that αGreedy may first pick all columns of group $\lceil d/\alpha \rceil$, then all columns of group $\lceil d/\alpha \rceil - 1$, and so on until group 1. Hence, the solution found by αGreedy consists of

$$\lceil d/\alpha \rceil! \times (\alpha H(\lceil d/\alpha \rceil - 1) + (d - \alpha(\lceil d/\alpha \rceil - 1))/\lceil d/\alpha \rceil)$$

$$= \lceil d/\alpha \rceil! \times (\alpha H(\lceil d/\alpha \rceil) + \frac{d}{\lceil d/\alpha \rceil} - \alpha)$$

columns, whereas the optimal solution contains $(\lceil d/\alpha \rceil)!$ columns. An example with $\alpha = 4, d = 10$ is shown in Figure 8.3. The example actually shows only half of the rows and columns. The first group is on the right etc. ■

8.3 HyperαGreedy algorithms and their worst case ratio.

For reasons explained in the introduction we are not satisfied with an αGreedy algorithm performing $O(m + n)$ iterations. To get rid of the $O(n)$ term, we now propose a modification of the αGreedy algorithm into a HyperαGreedy algorithm performing $O(m)$ iterations. In this section we describe the HyperαGreedy algorithm, and derive a bound on its worst case ratio.

This presentation parallels the presentation of αGreedy in Section 2. The differ-

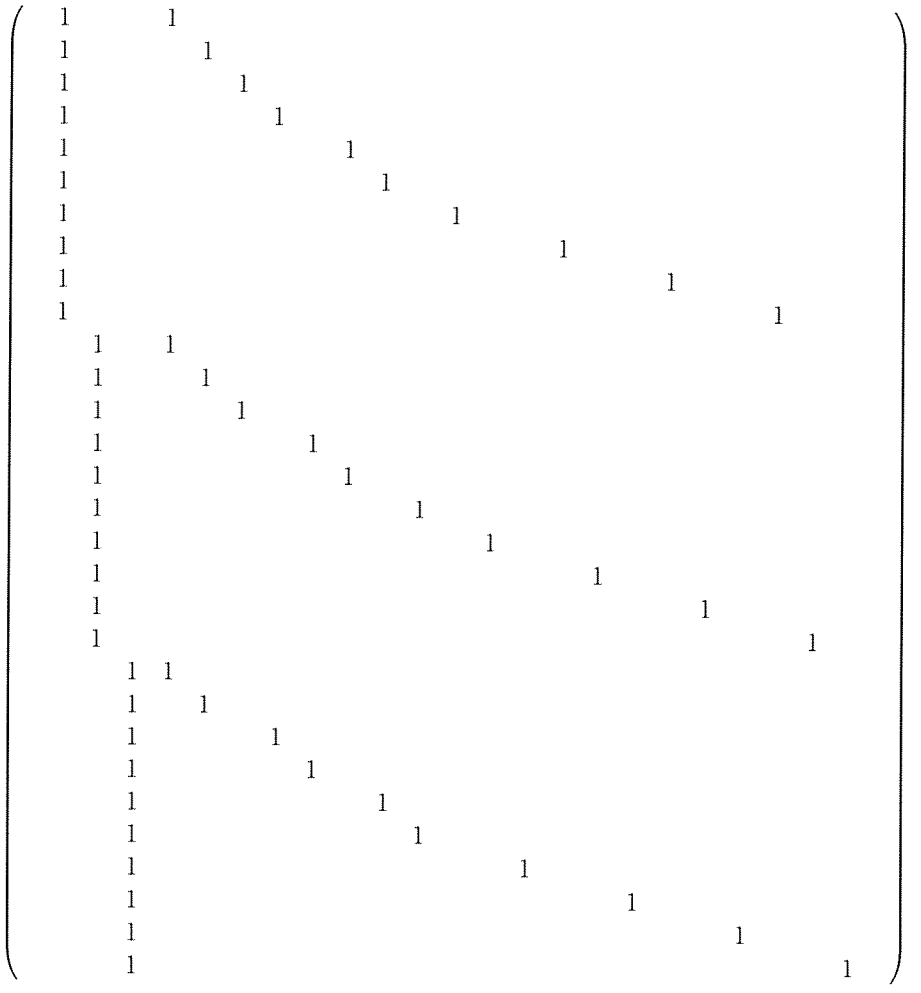

Figure 8.3: Tight worst case instance for Theorem 8.2

Algorithm 3 : HyperαGreedy

$x := 0$
$z := 0$
while $b \neq 0$ *do*
begin

select k *such that*
$c_k / \sum_{i=1}^{m} a_{ik} \leq \alpha \times \min_{1 \leq j \leq n}(c_j / \sum_{i=1}^{m} a_{ij})$

while for all i, $b_i > 0$ *do*
begin
$x_k := x_k + 1$
$b_i := \max(0, b_i - a_{ik})$ *for all* i
$z := z + c_k$
end
$a_{ij} := \min(a_{ij}, b_i)$ *for all* i, j
end.

Figure 8.4: Algorithm HyperαGreedy

ence between the two algorithms is that, in αGreedy, x_k stops increasing as soon as $b_i < a_{ik}$ for some i. Under the same conditions, x_k is increased by one more unit in HyperαGreedy. It is easily seen that the inner-while statement can be replaced by a statement of the form

$$x_k := x_k + \min_{1 \leq i \leq m} \lceil b_i / a_{ik} \rceil$$

with corresponding updates of b and z. (Contrast this with the statement $x_k := x_k + min_{1 \leq i \leq m} \lfloor b_i / a_{ik} \rfloor$ in the case of αGreedy.) This results in an algorithm performing $O(m)$ iterations, since HyperαGreedy covers then at least one b_i in each iteration of the outer while statement (i.e. satisfies at least one constraint).

We claim the following worst case ratio for HyperαGreedy :

Theorem 8.3 If x^* is an optimal solution of problem (P) and x' is the solution computed by HyperαGreedy, then

$$cx'/cx^* < 2\alpha H(A_{max}).$$

Proof. First, we introduce some notations. We view one execution of the outer while statement as a step of HyperαGreedy. Let ℓ be the number of such steps, and notice that ℓ is in $\{1,\ldots,m\}$. The data at the beginning of step r are denoted by A^r, b^r. Let k_r, $r \in \{1,\ldots,\ell\}$, be the column picked by HyperαGreedy in step r. Let i_r be any row such that

$$i_r = \arg \min_{1 \le i \le m} \left\lceil \frac{b_i^r}{a_{ik_r}^r} \right\rceil$$

and let

$$h_r = \left\lceil \frac{b_{i_r}^r}{a_{i_r k_r}^r} \right\rceil$$

Thus, h_r represents the increase of x_{k_r} in step r. Let

$$p_r = \left\lfloor \frac{b_{i_r}^r}{a_{i_r k_r}^r} \right\rfloor$$

Notice that $p_r \ge 1$ for all r in $\{1,\ldots,\ell\}$. We will use p^r in order to define a problem (P^1) that plays an important role in the proof. We define A_j^r to be the j-th column of A^r and β^r as follows:

$$\beta^r = \sum_{t=r}^{\ell} p_t A_{k_t}^t \qquad (r = 1,\ldots,\ell).$$

The idea behind this definition is that β^r is precisely the vector that would have been covered by αGreedy in steps $r,\ldots,\ell$, had αGreedy followed the same steps as HyperαGreedy. Notice that, in these steps, HyperαGreeedy covers b^r. This motivates the following lemma :

Lemma 8.4 For all r in $\{1,\ldots,\ell\}$ and i in $\{1,\ldots,m\}$, $\beta_i^r \le b_i^r$.

Proof. Fix r and i and assume that row i is covered by HyperαGreedy in step s, i.e. $i = i_s$. Observe that for $t > s$, $a_{ij}^t = 0$ for all j in $\{1,\ldots,n\}$. Now consider first the case where $r \le s$. Then, since $b_i^r - b_i^s$ is precisely the quantity 'covered' in steps $r, r+1,\ldots,s-1$, we have :

$$b_i^r = \sum_{t=r}^{s-1} h_t a_{ik_t}^t + b_i^s = \sum_{t=r}^{s-1} h_t a_{ik_t}^t + \frac{b_i^s}{a_{ik_s}^s} a_{ik_s}^s.$$

Since $h_t \ge p_t$ and $\dfrac{b_i^s}{a_{ik_s}^s} \ge p_s$, we obtain for $r \le s$,

$$b_i^r \ge \sum_{t=r}^{s} p_t a_{ik_t}^t = \sum_{t=r}^{\ell} p_t a_{ik_t}^t = \beta_i^r.$$

On the other hand, when $r > s$, $b_i^r = \beta_i^r = 0$. This proves Lemma 8.4. ∎

Define now a new problem (P^1) as follows :

$$\begin{array}{lll} \text{minimize} & cy & \\ \text{s.t.} & Ay & \geq \beta^1 \\ & y & \in \mathbb{Z}_+^n \end{array} \qquad (P^1)$$

Denote by y^* an optimal solution of (P^1), then,

Lemma 8.5 $cy^* \leq cx^*$.

Proof. Directly from Lemma 8.4. ∎

The next lemma makes more precise the intuitive interpretation of the vector β^1 that we gave earlier.

Lemma 8.6 When applied to (P^1), αGreedy may pick the sequence of columns $k_1, \ldots, k_\ell$ and increase y_{k_r} by p_{k_r} in step r, r in $\{1, \ldots, \ell\}$.

Proof. We prove this by induction on r. In fact we prove slightly more. For i in $\{1, \ldots, m\}$, j in $\{1, \ldots, n\}$ and r in $\{1, \ldots, \ell\}$, let

$$\alpha_{ij}^r = \min(a_{ij}, \beta_i^r)$$

We are going to prove that, when applied to the following problem (P^r) :

$$\begin{array}{lll} \text{minimize} & cy & \\ \text{s.t.} & \sum_{j=1}^n \alpha_{ij}^r y_j & \geq \beta_i^r \quad i \text{ in } \{1, \ldots, m\} \\ & y_j & \in \mathbb{Z}_+^n \end{array} \qquad (P^r)$$

αGreedy may pick column k_r in the first iteration; moreover, this column is equal to $A_{k_r}^r$, i.e.

$$\alpha_{ik_r}^r = a_{ik_r}^r \text{ for all } i \text{ in } \{1, \ldots, m\},$$

and, updating (P^r) yields (P^{r+1}). Lemma 8.6 directly follows from these claims.

To prove our claims, notice first that, for all i in $\{1, \ldots, m\}$, j in $\{1, \ldots, n\}$:

$$\alpha_{ij}^r = \min(a_{ij}, \beta_i^r) \leq \min(a_{ij}, b_i^r) = a_{ik_r}^r, \qquad (8.4)$$

(the inequality follows from Lemma 8.4). Moreover :

$$\alpha_{ik_r}^r = \min(a_{ik_r}, \beta_i^r) \geq \min(a_{ik_r}^r, \beta_i^r) = a_{ik_r}^r,$$

since $a_{ik_r} \geq a_{ik_r}^r$ and $\beta_i^r \geq a_{ik_r}^r$ by the definition of β^r. Together with (4) this implies $\alpha_{ik_r}^r = a_{ik_r}^r$ for all i in $\{1, \ldots, m\}$ as claimed.

Observing that column k_r of (P^r) is equal to $A^r_{k_r}$, while each other column j of (P^r) is smaller than or equal to A^r_j by (4), it is clear that αGreedy may pick column k_r in the first step (since HyperαGreedy did pick it.) For each i in $\{1,\ldots,m\}$

$$\frac{\beta^r_i}{\alpha^r_{ik_r}} = \sum_{t=r}^{\ell} \frac{p_t a^t_{ik_t}}{a^r_{ik_r}} = p_r + \sum_{t=r+1}^{\ell} \frac{p_t a^t_{ik_t}}{a^r_{ik_r}} \geq p_r. \tag{8.5}$$

Moreover, since row i_r was covered by HyperαGreedy applied to (P) in iteration r, $a^t_{i_r k_t} = 0$ for all t in $\{r+1,\ldots,\ell\}$. Thus

$$\frac{\beta^r_{i_r}}{\alpha^r_{i_r k_r}} = p_r. \tag{8.6}$$

From (5) and (6) it follows that, when applied to (P^r), αGreedy increases y_{k_r} by p_r in the first step, and decreases the right hand side by $p_r A^r_{k_r}$, i.e. β^r becomes β^{r+1}. Hence (P^r) is updated into (P^{r+1}). ∎

Let $y'_j = \sum_{\substack{r=1 \\ k_r=j}}^{\ell} p_r$. From the previous proof, it follows that $y' = (y'_1, y'_2, \ldots, y'_n)$ is an αGreedy solution to (P^1). Thus, by Theorem 8.1 and Lemma 8.5,

$$\begin{aligned} cy' &\leq \alpha H(\max_{1\leq j\leq n} \sum_{i=1}^{m} \alpha^1_{ij}) cy^* && \text{(by Theorem 8.1)} \\ &\leq \alpha H(\max_{1\leq j\leq n} \sum_{i=1}^{m} a_{ij}) cx^* && \text{(by Lemma 8.5)} \end{aligned}$$

Notice that the solution x' delivered by HyperαGreedy satisfies

$$x'_j = \sum_{\substack{r=1 \\ k_r=j}}^{\ell} h_r, \text{ and that } h_r \leq 2p_r \text{ for all } r \text{ in } \{1,\ldots,\ell\}.$$

Hence $cx' \leq 2cy' \leq 2\alpha\, H(A_{max}) cx^*$, as announced. ∎

It is not too difficult to see that the bound in Theorem 8.3 can not hold with equality. Indeed, if $x' = y'$, then the previous proof shows that $cx' \leq \alpha H(A_{max}) cx^*$. If $x' \neq y'$, then $b > \beta^1$, and hence $Ax^* > \beta^1$. The last part of the proof of Theorem 8.1, in Section 2, implies that, under these conditions, $cy' < \alpha H(A_{\max}) cx^*$, and thus $cx' \leq 2cy' < 2\alpha H(A_{max}) cx^*$.

In fact, we conjecture that the bound in Theorem 8.3 can be tightened as follows :

$$cx'/cx^* \leq 2\alpha \times H(d) \times \frac{A_{max}}{(A_{max}+1)}.$$

The following instance shows that this bound would be tight for all values of α and all integer values of A_{max} :

$$
\begin{array}{lccccccccc}
\text{Min} & \frac{d}{d\alpha}\, x_1 & +\frac{d}{(d-1)\alpha} x_2 & +\cdots+ & \frac{d}{2\alpha} x_{d-1} & +\frac{d}{\alpha} x_d & +x_{d+1} & & \\
\text{s.t.} & d\, x_1 & & & & & +x_{d+1} & \geq & d+1 \\
 & & d\, x_2 & & & & +x_{d+1} & \geq & d+1 \\
 & & & \ddots & & & \vdots & & \\
 & & & & d\, x_{d-1} & & +x_{d+1} & \geq & d+1 \\
 & & & & & d\, x_d & +x_{d+1} & \geq & d+1.
\end{array}
$$

For this instance, $A_{max} = d$. The optimal solution is given by $x_i^* = 0$ $(1 \leq i \leq d)$, $x_{d+1} = d+1$, and has value $d+1$. But HyperαGreedy could set $x_i = 2$ in iteration i $(1 \leq i \leq d)$, thus resulting in a solution with value $2\alpha d H(d)$.

8.4 Applications

The research in this chapter was motivated by our search for approximation algorithms for the Batch Selection and Job Grouping problem discussed in Chapter 7. Solving the Job Grouping Problem with the algorithm Greedy requires to select maximum cardinality groups. This column generation subproblem is known as the Batch Selection Problem. Chapter 7 shows that this Batch Selection Problem is NP-hard, and that all polynomial time approximation algorithms known to the authors have extremely bad worst case behavior. The αGreedy algorithm presented in this chapter however, only yields a polynomial time appromixation algorithm for Job Grouping with a 'good' worst case ratio, when a polynomial time approximation algorithm with good worst case ratio is available for Batch Selection. Hence, as yet the consequences of the result of the previous sections for the Job Grouping Problem are of limited interest. There are however, other applications of the results in the previous sections. We conclude this chapter with a brief discussion of two such applications.

The first application is to the Cutting Stock problem described in Section 1. Consider the rather general variant of this problem in which there is a cost associated with each raw. Then, the Cutting Stock problem can be modelled in the format *(P)*, where the cost c_j associated with the *j-th* cutting pattern is simply the cost of the corresponding raw. When this is the case, Gilmore and Gomory [1964] observed that the column generation subproblem reduces to a sequence of knapsack problems, one for each raw (see also Chvátal [1979]). Now, there is a simple approximation algorithm

for (the maximization version of) the knapsack problem with worst case ratio 1/2. This, together with Theorem 8.3, yields an approximation algorithm with worst case ratio $4H(A_{\max})$ for the Cutting Stock Problem. In fact, this ratio can even be made arbitrarily close to $2H(A_{\max})$ since the knapsack problem admits a fully polynomial time approximation scheme (see e.g. Nemhauser and Wolsey [1988]).

A second application arises when probabilistic logic is used to model uncertain information (e.g. the information contained in the knowledge base of an expert system). A fundamental problem in this framework is the Probabilistic Satisfiability problem, which can be informally stated as follows (see Nilsson [1986]) : given a set of propositional clauses and the probability that each clause is true, decide whether this probability assignment is consistent. We are now going to show how one variant of this problem can be modelled as a covering problem with a large number of columns (see also Kavvadias and Papadimitriou [1990], Hansen, Jaumard and Poggi de Aragão [1991]). To describe this model, let $\{C_1, \ldots, C_m\}$ be a set of clauses on the propositional variables $\{V_1, \ldots, V_n\}$, and let $p = (p_1, \ldots, p_m)$, where p_i is the probability assigned to clause $C_i, i = 1, \ldots, m$. Let $W = \{w_1, \ldots, w_{2^n}\} = \{True, False\}^n$ denote the set of possible worlds, i.e. the set of possible truth assignments for $\{V_1, \ldots, V_n\}$. We introduce now a $m \times 2^n$ matrix A, such that $a_{ij} = 1$ if w_j is a satisfying truth assignment for clause C_i, and $a_{ij} = 0$ otherwise. Then, the Probabilistic Satisfiability problem asks whether the following system has a feasible solution :

$$Ax \geq p, \sum_{i=1}^{2^n} x_i = 1, x \geq 0.$$

Assume now (without loss of generality for practical purposes) that all parameters p_i are rational numbers of the form $p_i = b_i/q$, with b_i and q integer $(i = 1, \ldots, m)$. Then it is easy to see that the above system is feasible if and only if the optimal value of the following linear programming problem is at most q :

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{2^n} x_i & \\ \text{s.t.} & Ax \geq b & (PS) \\ & x \in \mathbb{R}_+^{2^n}. & \end{array}$$

Kavvadias and Papadimitriou [1990] have observed that problem (PS) can be solved by a column generation approach, and that the column generation subproblem turns out in this case to be a weighted maximum satisfiability problem ($MAXSAT$). Since there exist polynomial time approximation algorithms with worst case ratio 3/4 for $MAXSAT$ (see e.g. Goemans and Williamson [1993]), Theorem 8.3 implies that one can find in polynomial time an integer-valued solution of (PS) whose value is at most $\frac{8}{3}H(m)$ times the optimal value of (PS). If this solution has value at most q, then it solves the Probabilistic Satisfiability problem. Otherwise, it could provide a reasonable initial solution (viz, set of columns) in order to solve (PS) by column generation.

Bibliography

Agnetis, A., 1989, No-wait flow shop scheduling with large lot size, Rap. 16.89, Universita degli studi di Roma 'La Sapienza'.

Agnetis, A., R.G. Askin, M.S. Sodhi, 1994, Tool addition strategies for flexible manufacturing systems, International Journal of Flexible Manufacturing Systems Vol. 6, 1994, pp. 287-310.

Agnetis, A., Lucertini, M., Nicolo, F., 1993, Flow management in flexible manufacturing cells with pipeline operations, Management Science Vol. 39, No. 3, pp. 294-306.

Ahmadi, R.H., 1993, A hierarchical approach to design, planning, and control problems in electronic circuit card manufacturing, in Perspectives in Operations Management, R.K. Sarin (Ed.), pp. 409-429, Kluwer Academic Publishers, Dordrecht, The Netherlands.

Ahmadi, J., Grotzinger, S., Johnson, D., 1988, Component allocation and partitioning for a dual delivery placement machine, Operations Research, Vol. 36, No. 2, pp. 176-191.

Ahmadi, R.H., P. Kouvelis, 1994, Staging problem of a dual delivery pick-and-place machine in printed circuit card assembly, Operations Research, Vol. 42, 1994, pp. 81-91.

Ahuja,R.K., Magnanti, T.L., Orlin, J.B., 1993, Network Flows, Prentice-Hall, Englewood Cliffs, New Jersey.

Armstrong, R., Lei, L., Gu, S., 1994, A bounding scheme for deriving the minimal cycle time of a single transporter N-stage process with time windows, European Journal of Operational Research 78, pp. 130-140.

Arora, S., Lund, C. Motwani, R. Sudan, M., Szegedy, M., 1992, On the intractability of approximation problems, Early draft, AT&T Bell Labs, NJ.

Askin, R.G., Dror, M., Vakharia, A.J., 1994, Printed circuit board family grouping and component allocation for a multimachine, open shop assembly cell, Naval Research Logistics, Vol. 41 pp. 587-608.

Asfahl, C.R., 1985, Robots and Manufacturing Automation, John Wiley & Sons, New

York, NY.

Balakrishnan, A., Vanderbeck F, 1993, A tactical planning model for mixed-model electronics assembly operations, CORE Discussion paper 9349, Catholic University of Louvain.

Ball, M.O., Magazine, M.J., 1988, Sequencing of insertions in printed circuit board assembly, Operations Research, Vol. 36, No. 2, pp. 192-201.

Bard, J.F., 1988, A heuristic for minimizing the number of tool switches on a flexible machine, IIE Transactions, Vol. 20, pp. 382-391.

Bard, J.F., Clayton, R.W., Feo, T.A., 1994, Machine setup and component placement in printed circuit board assembly, The International Journal of Flexible Manufacturing Systems, Vol. 6, No. 1, pp. 5-31.

Blazewicz, J., Ecker, K., Schmidt, G., Weglarz, J., 1993, Scheduling in computer and manufacturing systems, Springer-Verlag, Berlin Heidelberg.

Blazewicz, J., Eiselt, H.A., Finke, G., Laporte, G., Weglarz, J., 1991, Scheduling tasks and vehicles in a flexible maufacturing system, International Journal of Flexible Manufacturing Systems 4, pp. 5-16.

Blazewicz, J., Finke, G., 1994, Scheduling with resource management in manufacturing systems European journal of operational research, Vol. 76, pp. 1-14.

Burkard, R.E., Klinz, B., Rudolf, R., 1994, Perspectives of Monge properties in optimization, Research Report, TU Graz, Graz, Austria. (To appear in Discrete Applied Mathematics)

Carmon, T.F., Maimon, O.Z., Dar-el, E.M., 1989, Group set-up for printed circuit board assembly, International Journal of Production Research, Vol 27, pp. 1795-1810.

Chaillou, P., Hansen, P., Mahieu Y., 1989, Best network flow bounds for the quadratic knapsack problem, In: Combinatorial Optimization, B. Simeone (ed.) Lecture notes in Mathematics, Vol 1403. Springer Verlag, Berlin, pp. 225-235.

Chvátal, V., 1979, A greedy heuristic for the set-covering problem, Mathematics of Operations Research 4, pp. 233-235.

Chvátal, V., 1983, Linear Programming, Freeman, New York.

Cohen, G., Dubois, D., Quadrat, J.P., Viot, M., 1985, A linear-system-theoretic review

of discrete-event processes and its use for performance evaluation in manufacturing, IEEE Transactions on Automatic Control, Vol AC 30, No 3, pp. 210-220.

Cosmadakis, S.S., Papadimitriou, C.H., 1984, The traveling salesman problem with many visits to few cities, SIAM Journal on Computing, Vol. 13, No 1, pp. 99-108.

Cunninghame-Green, R., 1979, Minimax Algebra, Lecture Notes in Economics and Mahematical Systems, Springer Verlag, Berlin Heidelberg New York.

Crama, Y., 1995, Combinatorial models for production scheduling in automated manufacturing systems, 14th European Conference on Operational Research, Semi-plenary Papers, pp. 237-259.

Crama, Y., Flippo, O.E., Van de Klundert, J.J., Spieksma F.C.R. 1995a, The component retrieval problem in printed circuit board assembly, Working Paper, University of Limburg, Maastricht, The Netherlands.

Crama, Y., Flippo, O.E., Van de Klundert, J.J., Spieksma, F.C.R., 1995b, The assembly of printed circuit boards: a case with multiple machines and multiple board types, In revision for the European Journal of Operations Research.

Crama, Y., Kolen, A.W.J., Oerlemans, A.G., Spieksma, 1990, F.C.R., Throughput rate optimization in the automated assembly of printed circuit boards, Annals of Operations Research Vol. 26, pp. 455-480.

Crama, Y., Kolen, A.W.J., Oerlemans A.G., Spieksma F.C.R., 1994, Minimizing the number of tool switches on a flexible machine, International Journal of Flexible Manufacturing Systems 6, pp. 33-54.

Crama, Y., Oerlemans, A.G., 1994, A column generation approach to job grouping for flexible manufacturing systems, European Journal of Operation Research 78, 58-80.

Crama, Y., Oerlemans, A.G., Spieksma, F.C.R., 1994, Production Planning in Automated Manufacturing, Lecture Notes in Economics and Mathematical Systems 414, Springer-Verlag, Berlin, Germany.

Crama, Y., Klundert, J. J. van de, 1994, Approximation algorithms for integer covering problems via greedy column generation, RAIRO-Operations Research 28, pp. 283-302.

Crama, Y., Klundert, J. J. van de, 1995, Cyclic scheduling of identical parts in a robotic cell, To appear in Operations Research.

Crama, Y., Klundert, J. J. van de, 1996a, The optimality of short robot move cycles

in robotic cells, in preparation.

Crama, Y., Klundert, J. J. van de, 1996b, A disjunctive graph model for robot move seqeuencing problems, unpublished manuscript.

Dietrich, B.L., Lee ,J., Lee, Y.S., 1993, Order selection on a single machine with high set up costs, Annals of Operations Research 43, pp. 379-396.

Dobson, G., 1982, Worst-case analysis of greedy heuristics for integer programming with nonnegative data, Mathematics of Operations Research 7, pp. 515-531.

Drezner, Z., Nof, S., 1984, On optimizing bin picking and insertion plans for assembly robots, IIE Transactions, Vol. 16, 1984, 262–270.

Dyckhoff, H., 1990, A typology of cutting and packing problems, European Journal of Operational Research 44, pp. 145-159.

Foulds, L.R., Hamacher H.W., 1993, Optimal bin location and sequencing in printed circuit board assembly, European Journal of Operational Research Vol. 66, pp. 279–290.

Francis, R.L., Hamacher H.W., Lee C.-Y., Yeralan, S., 1994, Finding placement sequences and bin locations for cartesian robots, IIE Transactions, Vol. 26, pp. 47-59.

Gallo, G., Hammer, P.L., Simeone, B., 1980, Quadratic knapsack problems, Mathematical Programming Studies 12, pp. 132-149.

Garey, M.R., Johnson, D.S., 1979, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, New York.

Gilmore, P.C., Gomory, R.E., 1964, Sequencing a one state-variable machine: a solvable case of the traveling salesman problem, Operations research, Vol. 12, pp. 655-679.

Gilmore, P.C., Gomory, R.E., 1963, A linear programming approach to the cutting stock problem, Operations Research 9, pp. 849-859.

Gilmore, P.C., Lawler, E.L., Shmoys D.B., 1985, Well solved special cases, In Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds.) The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley-Interscience Series in Discrete Mathematics, John Wiley & Sons, Chichester New York Brisbane Toronto Singapore.

Goemans, M.X., Williamson, D.P., 1993, A new $\frac{3}{4}$—approximation algorithm for MAX

SAT, In Proc. of the third IPCO Conference, G. Rinaldi & L. Wolsey eds., pp. 313-321.

Goldschmidt, O., Hochbaum, D.S., Yu, G., 1992, Component assembly in the semiconductor industry : a study of covering in graphs and hypergraphs, Technical report ORP92-5, The University of Texas at Austin.

Goldschmidt, O., Nehme, D, Yu, G., 1993, On a generalization of the knapsack problem with applications to flexible manufactuirng systems and database partitioning, Working paper, University of Texas at Austin.

Granot, F., Skorin-Kapov, J., Tamir, A., 1993, Using quadratic programming to solve high multiplicity scheduling problems on parallel machines, Working Paper.

Gray, A.E., Seidmann, A., Stecke, K.E., 1993, A synthesis of decision models for tool management in automated manufacturing, Management Science, Vol. 39, pp. 549-567.

Haessler R.W., Sweeney, P.E., 1991, Cutting stock problems and solution procedures, European Journal of Operational Research 54, pp. 141-150.

Hall, N.G., Kamoun, H., Sriskandarajah, C., 1993, Scheduling in robotic cells: large cells. Working Paper, College of Business, The Ohio State University.

Hall, N.G., Kamoun, H., Sriskandarajah, C., 1995a, Scheduling in robotic cells: classification, two and three machine cells. Working Paper, College of Business, The Ohio State University. To appear in Operations Research.

Hall, N.G., Kamoun, H., Sriskandarajah, C., 1995b, Scheduling in robotic cells: complexity and steady state analysis . Working Paper, College of Business, The Ohio State University.

Hall, N.G., Potts, C.N., Sriskandarajah, C., 1995c, Parallel machine scheduling with a common server. Working Paper, College of Business, The Ohio State University

Hanen, C., Munier, A., 1994, Periodic scheduling of several hoists, In Proc. Fourth International Conf. Project Management and Scheduling, pp. 108-110.

Hansen, P., Jaumard, B., Poggi de Aragão, M., 1991, Column generation methods for probabilistic logic, ORSA Journal on Computing 3, 1991, pp. 135-148.

Hertz, A., 1995, private communication.

Hertz, A., Mottet, Y., Rochat, Y., 1996, On a scheduling problem in a robotized analytical system, Discrete Applied Mathematics 65, pp. 285-318.

Hochbaum, D.S., Shamir, R., 1991, Strongly polynomial algorithms for the high multiplicity scheduling problem, Operations Research Vol. 39, No. 4, pp. 648-653.

Horak, T., Francis, R.L., 1995, Utilization of machine characteristics in PC board assembly, Working paper, Rutgers University, Newark, New Jersey.

Ioachim, I., Soumis, F., 1995, Schedule eficiency in a robotic production cell, International Journal of Flexible Manufacturing Systems 7, pp. 5-26.

Jaikumar, R., 1986, Postindustrial Manufacturing, Harvard Business Review, pp 69-77.

Jaikumar, R., Wassenhove, L.N. van, 1989, A production planning framework for flexilbe manufacturing systems, JOurnal of manufacturing and operations management, Vol. 2, pp. 52-79.

Jeng, W.D., Lin, J.T., Wen, U.P., 1993, Algorithms for sequencing robot activities in a robot-centered parallel-processor workcell, Computers & Operations Research Vol. 20, No. 2, pp. 185-197.

Johnson, D.S., 1974, Approximation algorithms for combinatorial problems, Journal of Computer and System Sciences 9, pp. 256-278.

Johnson, D.S., 1974b, Worst-case behavior of graph coloring algorithms, In Proc. 5th Southeastern Conf. on Combinatorics, Graph Theory and Computing, pp. 513-527. Utilitas Mathematica Publ. Winnipeg, Ontario.

Kamoun, H., Hall., N.G., Sriskandarajah, C., 1993, Scheduling in robotic cells : heuristics and cell design, Working Paper, University of Toronto.

Kamoun, H., Sriskandarajah, C., 1993, The complexity of scheduling jobs in repetitive manufacturing systems, European Journal of Operational Research 70, pp. 350-364.

Karabati, S., Kouvelis, P., 1996, Cyclic scheduling in flow lines: modeling observations, effective heuristics and optimal cycle time minimization procedure, Naval Research Logistics 43, pp. 211-231.

Karp, R.M., 1978, A characterization of the minimum cycle mean in a digraph, Discrete Mathematics 23, pp. 309-311.

Kats, V.B., 1982, An exact optimal cyclic scheduling algorithm for multioperator service of a production line, Automation and Remote Control 42, No. 4 pt. 2, pp. 538-543.

Kats, V.B., 1995, private communication.

Kats, V.B., Levner, E.G., 1996, The constrained cyclic robotic flowshop problem: a solvable case. Proceedings of the international workshop on intelligent scheduling of robots and flexible manufacturing systems, Center for Technological Education Holon, Israel, pp. 115-128.

Kavvadias, D., Papadimitriou, C.P., 1990, A linear programming approach to reasoning about probabilities, Annals of Mathematics and Artificial Intelligence 1, 1990, pp. 189-205.

King, R.E., Hodgson, T.J., Chafee, F.W., 1993, Robot task scheduling in a flexible manufacturing cell. IIE Transactions Vol. 25, No. 2, pp. 80-87.

Kise, H, 1991, On an automated two-machine flowshop scheduling problem with infinite buffer, Journal of the Operations Research Society of Japan Vol. 34, No. 3, pp. 354-361.

Kise, H., Shioyama, T., Ibaraki, T., 1991, Automated two machine flowshop scheduling: a solvable case. IIE Transactions Vol. 23, No 1, pp. 10-16.

Klundert, J.J. van de, 1995, A note on the high multiplicity TSP, unpublished manuscript.

Kortsarz, G., Peleg, D., 1993, On choosing dense subgraphs. Extended abstracts, Dept. of Applied. Math. and Comp. Science., The Weizmann Institute, Rehovot, Israel.

Laarhoven, P.J.M. van, Zijm, W.H.M., 1993, Production preparation and numerical control in PCB assembly, International Journal of Flexible Manufacturing Systems, Vol. 5, No. 3, pp. 187–207.

Lieberman, R.W., Turksen, I.B., 1981, Crane scheduling problems. AIIE Transactions Volume 13, No 4, pp. 304-311.

Lei, L., 1993, Determining the optimal starting times in a cyclic schedule with a given route, Computers and Operations Research, Vol. 20, No. 8, pp. 807-816.

Lei, L., 1995, private communication.

Lei, L., Armstrong, R., Gu, S., 1993, Minimizing the fleet size with dependent time-window and single-track constraints, Operations Research Letters 14, pp. 91-98.

Lei, L., Wang, T.J., 1989, A proof : The cyclic hoist scheduling problem is NP-Complete. Working Paper 89-16, Graduate School of Management, Rutgers University.

Lei, L., Wang, T.J., 1991, The mimimum common cycle algorithm for cyclic scheduling of two hoists with time window constraints, Management Science Volume 37, No 12, pp. 1629-1639.

Lei, L., Wang, T.J., 1994, Determining optimal cyclic hoist schedules in a single-hoist electroplating line, IIE Transactions Vol. 26, No. 2, pp. 25-33.

Leipälä, T., Nevalainen, O., 1989, Optimization of the movements of a component placement machine, European Journal of Operational Research, Vol. 38, pp. 167-177.

Levner, E.G., 1995, private communication.

Levner, E.G., Kats, V.B., Meyzin, L.K., 1995, Fuzzy scheduling of robots in CIM environment. Working paper. Center for Technological Education Holon, Israel.

Levner. E., Kats, V.B., Levit, V.E., 1996, An improved algorithm for a cyclic robotic scheduling problem. Proceedings of the international workshop on intelligent scheduling of robots and flexible manufacturing systems, Center for Technological Education Holon, Israel, pp. 129-141.

Levner, E.G., Nemirovsky, A.S., 1994, A network flow algorithm for just in time project scheduling, European Journal of Operational Research, Vol. 79, pp. 167-175.

Lofgren, C.B., McGinnis, L.F., Tovey, C.A, 1991, Routing printed circuit cards through an assembly cell, Operations Research, Vol 39, pp. 992-1004.

Lovász, L., 1974, On the ratio of optimal integral and fractional covers, Discrete Mathematics 13, 1974, pp. 383-390.

Lund, C., Yannakakis, M., 1993, On the hardness of approximation minimization problems, Proc 25th ACM Symposium on the Theory of Computing, pp. 286-295.

McCormick, S.T., Pinedo, M.L., Shenker, S., Wolf, B., 1989, Sequencing in an assembly line with blocking to minimize cycle time, Operations Research Vol. 37, No 6, pp. 925-935.

McCormick, Rao, U.S., 1993, Some Complexity Results in Cyclic Scheduling, Working Paper, Faculty of Commerce, University of British Columbia, Vancouver, Canada.

Minoux, M., 1987, A class of combinatorial problems with polynomially solvable large scale set covering/partioning relaxations, R.A.I.R.O. Recherche opérationnelle/Operations Research 21, pp. 105-136.

Nemhauser, G.M., Wolsey, L.A., 1988, Integer and Combinatorial Optimization, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, New York Chichester Brisbane Toronto Singapore.

Nilsson, N.J., 1986, Probabilistic logic, Artificial Intelligence 28, pp. 71-87.

Pappadimitriou, C.H., Kanellakis, P.C., 1980, Flowshop scheduling with limited temporary storage, Journal of the ACM, Vol. 27, No 3, pp. 533-549.

Park, Y.B., 1994, Optimizing robot's service movement in a robot-centered FMC. Computers and Industrial Engineering Vol. 27, Nos 1-4, pp.47-50.

Philips, L.W., Unger P.S., 1976, Mathematical programming solution of a hoist scheduling program, AIIE Transactions, Vol. 8, No. 2, pp. 219-225.

Privault, C., Finke, G., 1993, Tool management on NC machines, Proc. Int. Conf. on Industr. Engineering and Prod. Management, Mons Belgium, pp. 667-676.

Rajagopalan, S., 1985., Scheduling problems in flexible manufacturing systems, Research Paper, Graduate school of Industrial Administration Carnegie-Mellon University, Pittsburgh, PA.

Rajagopalan, S., 1986, Formulation and heuristics solutions for parts grouping and tool loading in flexible manufacturing. In Stecke K.E. and Suri R. eds. Proceedings of the Second ORSA TIMS Conference on Flexible Manufacturing Systems. Elsevier Science Publishing B.V., Amsterdam, pp. 311-320.

Rock, H., 1984, The three-machine no-wait flowshop problem is NP-Complete, Journal of the ACM, Vol. 33, pp. 336-345.

Sethi, S.P., Sriskandarajah, C., Sorger, G., Blazewicz, J., Kubiak, W., 1992, Sequencing of parts and robot moves in a robotic cell. International Journal of Flexible Manufacturing Systems 4, pp. 331-358.

Shapiro G.W., Nuttle, H.L.W., 1988, Hoist scheduling for a PCB electroplating facility, IIE Transactions Vol. 20, pp.157-167.

Simon, H.U., 1988, The analysis of dynamic and hybrid channel assignment, Working Paper, Universität des Saarlandes, Saarbrücken, 1988.

Simon, H.U., 1990, On approximate solutions for combinatorial optimization problems, SIAM Journal on Discrete Mathematics 3, 1990, pp. 294-310.

Sriskandarajah, C., Hall., N.G., Kamoun, H., Wan, H., 1995, Scheduling large robotic cells, University of Toronto, Working Paper.

Song, W., Zabinsky, Z.B., Storch., R.L., 1993, An algorithm for scheduling a chemical processing tank line. Production Planning and Control Volume 4, No. 4, pp. 323-332.

Stecke, K.E., 1983, Formulation and solution of nonlinear integer production planning problems for flexible manufacturing systems. Management Science Vol. 29, No. 3, pp. 273-288.

Tang, C.S., 1988, A max-min allocation problem: Its solutions and applications, Operations Research, Vol. 36, No. 2, pp. 359-367 .

Tang, C.S., Denardo, E.V., 1988a, Models arising from a flexible manufacturing machine, part I: Minimization of the number of tool switches, Operation Research, Vol. 36, No. 5, pp. 767-777.

Tang, C.S., Denardo, E.V., 1988b, Models arising from a flexible manufacturing machine, part II: Minimization of the number of switches instants, Operation Research, Vol. 36, No. 5, pp. 778-784.

Viczián, I., 1993, Finding placement sequences and bin locations for cartesian robots, Working Paper, University of Würzburg.

Voogt, S., 1993, Short term scheduling in PCB assembly, Philips Report CTR 597-93-0106.

Walas, R.A., R.G. Askin, 1984, An algorithm for NC turret punch press tool location and hit sequencing, IIE Transactions, Vol. 16, pp. 280-287.

Younis, T.A., Cavalier T.M., 1990, On locating part bins in a constrained layout area for an automated assembly proces, Computers and Industrial Engineering, Vol. 18, pp. 111-118.

Whitney, C.K., Gaul T.S., 1985, Sequential decision procedures for batching and balancing in FMSs. Annals of Operations Research 3, pp. 301-316.

Samenvatting

Hoofdstuk 1 van dit proefschrift begint met een beschrijving van de achtergronden en motivatie van het onderzoek waarvan dit proefschrift verslag doet. Vervolgens bevat hoofdstuk 1 een inleiding in dit onderzoek en een korte samenvatting van de resultaten daarvan, die in de hoofdstukken 2 tot en met 8 in detail aan de orde komen. Deze samenvatting heeft dezelfde structuur.

Gedurende de afgelopen decennia hebben zich grote veranderingen voltrokken in zowel productietechnologie als de besturing van productiesystemen. De introductie van (volledig) geautomatiseerde machines, zoals robots, en geautomatiseerde planningssystemen heeft grote gevolgen gehad voor productiebedrijven in verscheidene bedrijfstakken. Daarnaast stelt de toegenomen internationale competitie hogere eisen aan producenten. De grote investeringen in geavanceerde productiemiddelen, die nodig zijn om concurrerend te blijven, maken het noodzakelijk dat deze productiemiddelen efficiënt worden ingezet. Aan de andere kant wordt het besturen en beheersen van geautomatiseerde systemen, opdat een hogere mate van efficiëntie inderdaad bereikt wordt, bemoeilijkt door de intrinsieke complexiteit van deze systemen.

Veel van de bestuurs- en beheersproblemen die zich binnen dergelijke systemen voordoen vallen onder de zogenaamde *deterministische scheduling problemen.* De theorie die voor dergelijke problemen tot nu toe ontwikkeld is, is echter niet toereikend om veel van de nieuwe problemen die zich specifiek in geautomatiseerde systemen voordoen goed op te lossen. Aan de andere kant is het vakgebied van de Combinatorische Optimalisering in staat gebleken om de aard van verwante problemen goed te doorgronden en in geautomatiseerde oplossingsprocedures voor dergelijke problemen te voorzien. In dit proefschrift onderzoeken we combinatorische eigenschappen van, en geautomatiseerde oplossingsmethoden voor, productieplanningsproblemen die zich voordoen in geautomatiseerde productie systemen.

Er bestaat een grote variëteit aan planningsproblemen in geautomatiseerde productie systemen. In het onderzoek waarvan dit proefschrift verslag doet, hebben we ons derhalve moeten beperken. We hebben het onderzoek beperkt tot drie uiteenlopende deelgebieden, en wel robotic cells, printed circuit board assembly, en tool management.

Een *robotic cell* is een productie systeem dat bestaat uit een aantal machines, een invoer buffer, een uitvoer buffer, en een robot. We beschouwen in dit proefschrift met name robotic flowshops. In een *robotic flowshop*, komen de (onderdelen van) producten die bewerkingen moeten ondergaan in de cell beschikbaar bij de invoer buffer. Daarna moeten zij bewerkingen ondergaan op ieder van de machines, in een gegeven, en voor alle producten identieke, volgorde. Als de producten alle bewerkingen hebben ondergaan kunnen zij worden afgeleverd bij de uitvoer buffer. Het transport van de

producten tussen de machines en de buffers wordt uitgevoerd door de robot. Planning problemen die zich voordoen in robotic flowshops hebben vaak tot doel de bereikte productiviteit te maximaliseren. Deze productiviteit hangt af van de bewerkingstijden van de producten, de transporttijden van de robot, en de interactie tussen de activiteiten van de robot en de machines. In het algemeen wordt in robotic flowshop planningsproblemen gevraagd een volgorde te specificeren waarin de producten van de invoer buffer worden gepakt door de robot, alsmede een volledige volgorde van robot bewegingen te specificeren.

Deel 1 van dit proefschrift betreft robotic cells en beslaat de hoofdstukken 2,3, en 4. In hoofdstuk 2 geven we een overzicht van onderzoek dat is verricht naar dergelijke planningsproblemen in robotic cells en in het bijzonder robotic flowshops. Daarnaast leggen we enkele nog niet eerder opgemerkte verbanden tussen resultaten van diverse onderzoekers. In hoofdstuk 2 identificeren we bovendien enkele belangrijke open problem met betrekking tot het plannen van robotic cells. Voor enkele van die problemen geven we reeds in hoofdstuk 2 een oplossing, andere komen in de hoofdstukken 3 en 4 aan de orde. In hoofdstuk 2 tonen we aan dat een algemene versie van het probleem NP-moeilijk is, hetgeen inhoudt dat het onwaarschijnlijk is dat dit probleem in het algemeen op een efficiënte wijze exact kan worden opgelost. In deze algemene versie moeten zowel de volgorde waarin de producten van de invoer buffer worden gepakt, alsmede de volgorde waarin de robot haar transport activiteiten verricht, worden bepaald. Zelfs als beide volgordes zijn vastgelegd, is het echter niet à priori duidelijk hoe het meest efficiente productie plan eruit ziet. In hoofdstuk 2 geven we, voor een bepaald type robotic flowshop, een efficient algoritme voor het bepalen van zo'n productie plan. Dit algoritme is algemener en efficienter dan verscheidene eerder gepubliceerde algoritmen.

In hoofdstuk 3 beschouwen we voor hetzelfde type robotic flowshop, het probleem van het bepalen van een korte optimale robot activiteiten volgorde in het geval alle producten identiek zijn (de volgorde waarin de producten van de invoerbuffer worden gepakt doet er dan niet toe). We laten zien dat zelfs voor robotic flowshops met veel machines, dit probleem efficient kan worden opgelost. Onze oplossingsmethode maakt, verassenderwijs, gebruikt van het concept van zogenaamde pyramidale permutaties, die eerder zijn onderzocht in verband met het wel bekende Handelsreiziger probleem.

In hoofdstuk 4 beschouwen we een vermoeden van Sethi et al. [1992] dat stelt dat in een bepaald type robotic flowshop, de meest efficiente productie wijze kan worden gerealiseerd met korte robot activiteiten volgordes die de robot herhaaldelijk uitvoert. Dergelijke korte volgordes zijn aantrekkelijk vanuit zowel een plannings- als een beheersoogpunt. We laten zien dat een nauwelijks zwakkere versie van dit vermoeden inderdaad juist is.

Deel 2 van dit proefschrift heeft betrekking op de assemblage van printed circuit boards. We beschouwen twee praktijk problemen. De assemblage van printed circuit boards behelst onder andere het plaatsen van componenten op lege printed circuit boards. De effiëntie waarmee dit plaatsen van componenten wordt uitgevoerd kan op de volgende twee manieren worden beïnvloed. De componenten worden door een gripper gepakt van feeders die aan een feeder rack worden bevestigd. Iedere feeder bevat

componenten van een enkel type. Tussen het pakken van opeenvolgende componenten moet het rack worden verplaatst, zodanig dat een feeder met componenten van het juiste type onder de gripper wordt gepositioneerd. De tijdsduur van dergelijke verplaatsingen hangt af van de lengte van de verplaatsing van het rack. De tijdsduur van de assemblage kan dus worden beïnvloed door de volgorde waarin de componenten gepakt worden, en door het toewijzen van feeders aan posities van het tape rack.

De tijdsduur van de assemblage wordt ook nog beinvloed door de afstand tussen locaties op het printed circuit board behorende bij componenten die direct na elkaar geplaatst dienen te worden.

Als zowel de volgorde waarin de componenten dienen te worden gepakt, als een toewijzing van feeders aan het rack gegeven zijn, doet zich nog het volgende probleem voor. In geval voor een bepaald type component meer dan één feeder aan het rack is toegewezen die componenten van dit type bevat, kan men, voor ieder component van dit type dat gepakt moet worden, kiezen van welke feeder de gripper deze component pakt. Het probleem dat er in bestaat deze keuze zo te maken dat de assemblage zo snel mogelijk verloopt, heet het Component Retrieval probleem. De complexiteit van dit probleem hangt af van de exacte werking van de assemblage machines. Voor de machines die we in hoofdstuk 5 beschouwen, beweren Bard et al. [1993] dat dit probleem eenvoudig als een kortste pad probleem is te modeleren. We laten in hoofdstuk 5 zien dat hun oplossingsmethode onjuist is en geven een alternatieve, en correcte, oplossingsmethode.

In hoofdstuk 6 beschouwen we het probleem van het efficient produceren van een aantal printed circuit boards van verschillende types op een aantal machines. Het bestuderen van deze problemen is het gevolg van een samenwerkingsproject met het Center for Manufacturing Technology van Philips Nederland N.V. De probleem instanties die we oplossen zijn door hen ter bechikking gesteld. In de oplossingsmethode die door Philips word gehanteerd, worden de borden samengevoegd tot een imaginair, samengesteld board. Uit een goed assemblage plan voor dit samengestelde board worden dan, hopelijk ook goede, assemblage plannen voor de oorspronkelijke boards afgeleid. In de in dit proefschrift beschreven oplossingsmethode hebben we daarentegen geprobeerd juist zoveel mogelijk rekening te houden met de individuele bord karakteristieken. De juistheid van deze keuze blijkt uit onder andere uit de kwaliteit van de oplossingen, die ten opzichte van de oplossingen van Philips een verbetering van ongeveer 17 % opleveren. Daarnaast hebben we kunnen aantonen dat een verbetering van meer dan ongeveer 25 % niet mogelijk is.

Het derde deel van dit proefschrift tenslotte, heeft betrekking op tool management. Een van de belangrijkste karakteristieken van moderne, geautomatiseerde, productie systemen is flexibiliteit. Een van de verschijningsvormen van flexibiliteit is de zogenaamde machine flexibiliteit, i.e. het vermogen van machines om met relatief kleine omsteltijden een grote varieteit aan productiehandelingen te verrichten. Deze machine flexibiliteit is grotendeels het gevolg van de mogelijkheid die moderne machines hebben om, al naar gelang de te verrichten handeling, gebruik te maken van verschillende tools, gereedschappen. Het wisselen van tools kost weinig tijd als de benodigde tools

in het tool magazijn van de machine aanwezig zijn. Als de benodigde tools echter van buiten de machine moeten komen, nemen de omsteltijden aanmerkelijk toe. Het wordt duidelijk dat de efficiëntie van de productie beïnvloed kan worden door de toewijzing (over tijd) van tools aan machines.

Toewijzingsproblemen die zich in dit kader voordoen zijn in de literatuur veelvuldig onderzocht. Vooral groeperingsproblemen hebben veel aandacht gekregen. Een group of batch van parts is een verzameling van producten waarvoor geldt dat de tools die nodig zijn voor de productiehandelingen die zij moeten ondergaan, gezamenlijk in het tool magazijn passen. Een veel voorkomend probleem is het zogenaamde Job Grouping probleem : is het mogelijk de verzameling van alle te produceren parts op te splitsen in (maximaal) zoveel batches als er machines beschikbaar zijn. Een manier om dit probleem op te lossen is als volgt. Genereer herhaaldelijk zo groot mogelijke batches, totdat alle parts zijn toegewezen aan een machine. Het probleem van het vinden van een zo groot mogelijke batch heet het Batch Selection probleem.

In hoofdstuk 7 beschouwen we het Job Grouping probleem, het Batch Selection probleem en enkele gerelateerde problemen. het is bekend dat voor deze problemen waarschijnlijk geen efficiente exacte oplossingsmethode bestaan. In hoofdstuk 7 onderzoeken we de mogelijkheid van het bestaan van oplossingsmethoden met een gegarandeerd goed gedrag. Daarmee bedoelen we dat de oplossingen die door deze oplossingsmethoden worden gegeven niet meer dan bijvoorbeeld een bepaald percentage verschillen van de optimale oplossing. We tonen in hoofdstuk 7 echter aan dat alle ons bekende oplossingsmethoden uit de literatuur zich arbitrair slecht kunnen gedragen. Aan de andere kant geven we enkele negatieve resultaten : voor bepaalde garanties kunnen we aantonen dat oplossingsmethode met die garanties waarschijnlijk niet kunnen bestaan.

In hoofdstuk 8 beschouwen we paren van problemen die gerelateerd zijn zoals het Job Grouping probleem en het Batch Seclection probleem. Dus, meer algemeen, (master-slave) paren van problemen waarvoor geldt dat een oplossing gevonden kan worden voor het master probleem, door het herhaaldelijk oplossen van een slave probleem. We vragen ons af wat de relatie is tussen de garanties van oplossingen voor beide problemen. In het bijzonder vragen we ons het volgende af. Stel dat we een oplossing voor het master probleem verkrijgen door het herhaaldelijk toepassen van een oplossingsmethode voor het slave probleem met een bepaalde garantie. Welke garantie kunnen we nu geven voor de zo verkregen oplossing voor het master probleem? In hoofdstuk 8 geven we scherpe karakteriseringen van de relatie tussen beide garanties.

Curriculum vitae

Joël Joris van de Klundert werd in 1967 geboren te Losser. Van 1979 tot 1985 volgde hij onderwijs aan het R.K. Lyceum De Grundel te Hengelo (ov). In 1985 behaalde hij zijn Gymnasium β diploma en begon een studie Bestuurlijke Informatica aan de Faculteit der Economische Wetenschappen van de Erasmus Universiteit Rotterdam. Tijdens zijn studie was hij gedurende twee jaar werkzaam als student-assistent bij de vakgroep Informatica. In 1991 studeerde hij af; zijn doctoraal scriptie schreef hij over een onderzoek betreffende de parallelle complexiteit van enkele locatie problemen. In 1992 trad hij als assistent in opleiding in dienst bij de sectie Besliskunde van de Faculteit der Economische Wetenschappen en Bedrijfskunde van de Rijksuniversiteit Limburg. Gedurende 1995 werd deze aanstelling een half jaar onderbroken voor een aanstelling als toegevoegd docent bij dezelfde sectie.